

eCosPro® Reference Manual

eCosPro® Reference Manual

Copyright © 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

Documentation licensing terms

This file is part of eCosPro®. Distribution of the work or derivative of the work in any form is prohibited unless prior permission obtained from the copyright holder.

Trademarks

Altera® and Excalibur™ are trademarks of Altera Corporation.

AMD® is a registered trademark of Advanced Micro Devices, Inc.

ARM®, StrongARM®, Thumb®, ARM7™, ARM9™ is a registered trademark of Advanced RISC Machines, Ltd.

Cirrus Logic® and Maverick™ are registered trademarks of Cirrus Logic, Inc.

Cogent™ is a trademark of Cogent Computer Systems, Inc.

Compaq® is a registered trademark of the Compaq Computer Corporation.

eCos®, eCosCentric® and eCosPro® are registered trademarks of eCosCentric Limited.

Fujitsu® is a registered trademark of Fujitsu Limited.

IBM®, and PowerPC™ are trademarks of International Business Machines Corporation.

IDT® is a registered trademark of Integrated Device Technology Inc.

Intel®, i386™, Pentium®, StrataFlash® and XScale™ are trademarks of Intel Corporation.

Intrinsyc® and Cerf™ are trademarks of Intrinsyc Software, Inc.

Linux® is a registered trademark of Linus Torvalds.

Matsushita™ and Panasonic® are trademarks of the Matsushita Electric Industrial Corporation.

Microsoft®, Windows®, Windows NT® and Windows XP® are registered trademarks of Microsoft Corporation, Inc.

MIPS®, MIPS32™ MIPS64™, 4K™, 5K™ Atlas™ and Malta™ are trademarks of MIPS Technologies, Inc.

Motorola®, ColdFire® is a trademark of Motorola, Inc.

NEC® V800™, V850™, V850/SA1™, V850/SB1™, VR4300™, and VRC4375™ are trademarks of NEC Corporation.

PMC-Sierra® RM7000™ and Ocelot™ are trademarks of PMC-Sierra Incorporated.

Red Hat, RedBoot™, GNUPro®, and Insight™ are trademarks of Red Hat, Inc.

Samsung® and CalmRISC™ are trademarks or registered trademarks of Samsung, Inc.

Sharp® is a registered trademark of Sharp Electronics Corp.

SPARC® is a registered trademark of SPARC International, Inc., and is used under license by Sun Microsystems, Inc.

Sun Microsystems® and Solaris® are registered trademarks of Sun Microsystems, Inc.

SuperH™ and Renesas™ are trademarks owned by Renesas Technology Corp.

Texas Instruments®, OMAP™ and Innovator™ are trademarks of Texas Instruments Incorporated.

Toshiba® is a registered trademark of the Toshiba Corporation.

UNIX® is a registered trademark of The Open Group.

All other brand and product names, trademarks, and copyrights are the property of their respective owners.

Warranty

eCos and RedBoot are open source software, covered by a modified version of the GNU General Public Licence (<http://www.gnu.org/copyleft/gpl.html>), and you are welcome to change it and/or distribute copies of it under certain conditions. See <http://ecos.sourceforge.org/license-overview.html> for more information about the license.

eCos and RedBoot software have NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

Other copyrights

Documentation on the lwIP TCP/IP stack includes portions derived from documentation distributed with the following license:

- * Copyright (c) 2001 Swedish Institute of Computer Science.
- * Redistribution and use in source and binary forms, with or without modification,
- * are permitted provided that the following conditions are met:
- *
- * 1. Redistributions of source code must retain the above copyright notice,
- * this list of conditions and the following disclaimer.
- * 2. Redistributions in binary form must reproduce the above copyright notice,
- * this list of conditions and the following disclaimer in the documentation
- * and/or other materials provided with the distribution.
- * 3. The name of the author may not be used to endorse or promote products
- * derived from this software without specific prior written permission.
- *
- * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AS IS” AND ANY EXPRESS OR IMPLIED
- * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
- * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
- * SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
- * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
- * OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
- * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
- * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
- * IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
- * OF SUCH DAMAGE.

Table of Contents

I. Atmel AT45xxxxx DataFlash Device Driver	xvii
Overview	19
Instantiating a DataFlash Device	21
II. SST 39VFXXX Flash Device Driver	xxv
Overview	27
Instantiating an 39vfxxx Device	29
III. SMSC LAN9118 Ethernet Driver	xxxix
SMSC LAN9118 Ethernet Driver	41
IV. ST M48T35A Wallclock Device Driver	xlvi
ST M48T35A Wallclock Device Driver	47
V. lwIP - the lightweight IP stack for eCosPro®	xlix
1. lwIP overview	1
Introduction	1
lwIP sources and ports	1
External documentation	2
Licensing	2
2. Basic concepts	3
Structure	3
Application Programming Interfaces (APIs)	3
Protocol implementations	3
Packet data buffers	4
Configurability	4
Limitations	5
Quick Start	6
3. Port	9
Port status	9
4. Sequential API	11
Overview	11
Comparison with BSD sockets	11
Netbufs	11
TCP/IP thread	11
Usage	12
API declarations	12
Types	12
IP address representation	12
Error codes	14
API reference	14
netbuf_new()	15
netbuf_delete()	17
netbuf_alloc()	19
netbuf_free()	21
netbuf_ref()	23
netbuf_len()	25
netbuf_data()	27

netbuf_next()	29
netbuf_first()	31
netbuf_copy()	33
netbuf_copy_partial()	35
netbuf_chain()	37
netbuf_fromaddr()	39
netbuf_fromport()	41
netconn_new()	43
netconn_new_with_callback()	45
netconn_new_with_proto_and_callback()	47
netconn_delete()	49
netconn_type()	51
netconn_peer()	53
netconn_addr()	55
netconn_bind()	57
netconn_connect()	59
netconn_disconnect()	61
netconn_listen()	63
netconn_accept()	65
netconn_recv()	67
netconn_write()	69
netconn_send()	71
netconn_close()	73
netconn_err()	75
5. Raw API	77
Overview	77
Usage	77
Callbacks	78
tcp_arg()	78
TCP connection setup	79
tcp_new()	79
tcp_bind()	81
tcp_listen()	83
tcp_accept()	85
tcp_connect()	87
Sending TCP data	89
tcp_write()	89
tcp_sent()	91
Receiving TCP data	93
tcp_recv()	93
tcp_recved()	95
Application polling	97
tcp_poll()	97
Closing connections, aborting connections and errors	99
tcp_close()	99
tcp_abort()	101
tcp_err()	103
Lower layer TCP interface	105

UDP interface	105
udp_new()	105
udp_remove()	107
udp_bind()	109
udp_connect()	111
udp_disconnect()	113
udp_send()	115
udp_recv()	117
System initialisation	119
VI. Object Loader	121
Object Loader	123
Extending the Object Loader	129
VII. The eCos NAND Flash Library	133
6. NAND Library Overview	135
Description	135
Structure of the library	135
Device support	136
Danger, Will Robinson! Danger!	136
Differences between NAND and NOR flash	136
Preparing for deployment	138
7. Using the NAND library	139
Configuring the NAND library	139
The NAND Application API	140
Device initialisation and lookup	140
NAND device addressing	141
NAND device partitions	141
About the spare area	142
Manipulating the NAND array	142
Reading data	142
Writing data	143
Erasing blocks	144
Common error returns	145
Ancillary NAND functions	145
8. Writing NAND device drivers	147
Planning a port	147
Driver structure and layout	147
Chip partitions	147
Locking against concurrent access	148
Required CDL declarations	148
High-level (chip) functions	148
Device initialisation	149
Reading, writing and erasing data	150
Searching for factory-bad blocks	151
Declaring the function set	151
Low-level (board) functions	152
Talking to the chip	152
Setting up the chip partition table	153

Putting it all together.....	153
ECC implementation	154
The ECC interface.....	154
9. Tests and utilities.....	157
Unit and functional tests.....	157
Ancillary NAND utilities.....	158
10. Samsung K9 family NAND chips	159
Overview	159
Using this driver in a board port.....	159
Memory usage	160
Low-level functions required from the platform HAL	160
11. ST Microelectronics NANDxxxx3a chips.....	163
Overview	163
Using this driver in a board port.....	163
Memory usage note	164
Low-level functions required from the platform HAL	164
VIII. Synthetic Target NAND Flash Device.....	167
Synthetic Target NAND Flash Device	169
IX. NXP LPC2xxx variant HAL.....	179
Overview	181
On-chip subsystems and peripherals.....	183
The HAL Port.....	187
X. Keil MCB2387 Board Support	189
Overview	191
Setup.....	193
Configuration	195
The HAL Port.....	201
XI. Phytex phyCORE LPC2294 Board Support.....	203
Overview	205
Setup.....	207
Configuration	211
The HAL Port.....	215
XII. Ashling EVBA7 Eval Board Support.....	217
Overview	219
Setup.....	221
Configuration	223
The HAL Port.....	225
XIII. Embedded Artists QuickStart Board Support.....	227
Overview	229
Setup.....	231
Configuration	235
The HAL Port.....	237

XIV. IAR KickStart Card Support	243
Overview	245
Setup	247
Configuration	251
The HAL Port	253
XV. Embedded Artists LPC2468 OEM Board Support	259
Overview	261
Setup	263
Configuration	267
The HAL Port	273
XVI. ST STR7XX variant HAL	275
Overview	277
On-chip Subsystems and Peripherals	279
The HAL Port	281
Power Management	283
XVII. STR7XX ADC Driver	289
STR7XX ADC Driver	291
XVIII. ST STR710-EVAL Board HAL	293
Overview	295
Setup	297
Configuration	305
JTAG debugging support	309
The HAL Port	311
XIX. Atmel AT91 Processor Variant Support	313
Overview	315
Hardware definitions	317
Interrupt Controller	319
Timers	321
Serial UARTs	323
XX. Atmel AT91SAM7 Processor Variant Support	325
Overview	327
Hardware definitions	329
Interrupt Vector Definitions	331
XXI. Atmel AT91SAM7A2-EK Board Support	335
Overview	337
Setup	339
Configuration	345
JTAG debugging support	347
The HAL Port	349
XXII. Atmel AT91SAM7A3-EK Board Support	353
Overview	355
Setup	357
Configuration	361
JTAG debugging support	365
The HAL Port	367

XXIII. Atmel AT91SAM7S-EK Board Support	371
Overview	373
Setup.....	375
Configuration	383
JTAG debugging support.....	387
The HAL Port.....	391
XXIV. Atmel AT91SAM7X-EK Board Support.....	395
Overview	397
Setup.....	399
Configuration	407
JTAG debugging support.....	411
The HAL Port.....	415
XXV. Atmel SAM9 Processor Support.....	419
Overview	421
Hardware definitions	423
Interrupt controller	425
Timers.....	431
Serial UARTs	433
Multimedia Card Interface (MCI) driver	435
Two-Wire Interface (TWI) driver.....	437
Power saving support	439
XXVI. Atmel AT91SAM9260 Evaluation Kit Board Support.....	441
Overview	443
Setup.....	445
Configuration	451
JTAG debugging support.....	455
The HAL Port.....	457
XXVII. Atmel AT91SAM9261 Evaluation Kit Board Support	463
Overview	465
Setup.....	467
Configuration	473
JTAG debugging support.....	477
The HAL Port.....	479
XXVIII. Atmel AT91SAM9263 Evaluation Kit Board Support.....	485
Overview	487
Setup.....	489
Configuration	495
JTAG debugging support.....	499
The HAL Port.....	501
XXIX. Atmel AT91RM9200 Processor Support	507
Overview	509
Hardware definitions	511
Interrupt controller	513
Timer counters.....	517
Serial UARTs	519

Multimedia Card Interface (MCI) driver	521
Two-Wire Interface (TWI) driver	523
Power saving support	525
XXX. Atmel AT91RM9200 Development Kit/Evaluation Kit Board Support	527
Overview	529
Setup	531
Configuration	539
JTAG debugging support	543
The HAL Port	545
XXXI. Cogent CSB337 Board Support	551
Overview	553
Setup	555
Configuration	559
The HAL Port	561
XXXII. KwikByte KB920x Board Family Support	563
Overview	565
Setup	567
Configuration	575
The HAL Port	579
XXXIII. SSV DNP/9200 with DNP/EVA9 Board Support	583
Overview	585
Setup	587
Configuration	595
JTAG debugging support	599
The HAL Port	601
XXXIV. Motorola MX1ADS/A Board Support	607
Overview	609
Setup	611
Configuration	617
The HAL Port	621
XXXV. ARM Versatile 926EJ-S Board Support	623
Overview	625
Setup	627
Configuration	631
The HAL Port	633
XXXVI. Intel XScale IXP4xx Network Processor Support	635
Overview	637
IXP4xx hardware definitions	639
IXP4xx interrupt controller	641
General-purpose timers	645
Watchdog	647
Serial UARTs	649
PCI bus controller	651
PCI bus IDE controllers	653
CompactFlash cards in TrueIDE mode	655

GPIO	657
XXXVII. Intel IQ80321 Board Support	659
Overview	661
Setup	663
Configuration	673
The HAL Port.....	675
XXXVIII. FAT File System Support	677
12. Introduction	679
13. Configuring the FAT Filesystem	681
Including FAT Filesystem in a Configuration.....	681
Configuring the FAT Filesystem.....	682
14. Using the FAT Filesystem	685
15. Removable Media Support.....	687
16. Non-ASCII Character Set Support.....	689
17. Testing.....	691
XXXIX. Journalling Flash File System v2 (JFFS2).....	693
Journalling Flash File System v2 overview	695
Using JFFS2	697
XL. Disk IO Package	703
18. Introduction	705
19. Configuring the DISK I/O Package.....	707
Including DISK I/O in a Configuration.....	707
Configuring the DISK I/O Package.....	707
20. Usage.....	709
21. Hardware Driver Interface.....	711
DevTab Entry	711
Disk Controller Structure	712
Disk Channel Structure.....	712
Disk Functions Structure	713
Callbacks	715
Putting It All Together.....	717
XLI. iPAQ Framebuffer Device Driver.....	719
iPAQ Framebuffer Device Driver.....	721
XLII. CSB337/900 Framebuffer Device Driver	723
CSB337/900 Framebuffer Device Driver.....	725
XLIII. Dallas DS1302 Wallclock Device Driver	727
Dallas DS1302 Wallclock Device Driver.....	729
XLIV. Dallas DS1306 Wallclock Device Driver	731
Dallas DS1306 Wallclock Device Driver.....	733
XLV. Dallas DS1390 Wallclock Device Driver	735
Dallas DS1390 Wallclock Device Driver.....	737
XLVI. Intersil ISL12028 Wallclock Device Driver	739
Intersil ISL12028 Wallclock Device Driver.....	741

XLVII. eCosPro™ Standard C++ library support package	743
22. Introduction	745
Overview of features	745
23. Usage	747
Requirements	747
Issues to consider	747
Using C++ exceptions	747
Application size	748
C++ exceptions in callbacks	748
Licensing	749
Open issues	749
GCC 3.3.x issues	749
GCC 3.4.x issues	749
Generic issues	750
24. Testing	751
25. Toolchain	753
XLVIII. gcov Test Coverage Support	755
Test Coverage	757
XLIX. Robust Boot Loader	765
Robust Boot Loader	767
RedBoot Commands	771
Application Library	775
Application Library Extensions	779
L. RedBoot Extra Initialization	783
RedBoot Extra Initialization	785
LI. ecoflash Flash Programming Utility	787
ecoflash Flash Programming Utility	789
LII. Flash Safe	799
Flash Safe	801
Flash Safe Programmer Interface	803

List of Tables

4-1. lwIP sequential API error codes 14

List of Examples

4-1. This example shows the basic mechanisms for using netbufs..... 17

4-1. This example shows a simple use of the netbuf_ref() 23

4-1. This example shows how to use the netbuf_next() function..... 29

4-1. This example shows a simple use of netbuf_copy() 33

4-1. This example shows how to open a TCP server on port 2000..... 65

4-1. This example demonstrates usage of the netconn_recv() function 67

4-1. This example demonstrates basic usage of the netconn_write() function 69

4-1. This example demonstrates basic usage of the netconn_send() function 71

1. Mounting and unmounting a JFFS2 filesystem 697

2. Secure erase usage 701

I. Atmel AT45xxxxxx DataFlash Device Driver

Overview

Name

Overview — eCos Support for Atmel AT45xxxxxx DataFlash Devices and Compatibles

Description

The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` Atmel AT45xxxxxx V2 flash driver package implements support for the AT45xxxxxx family of flash devices and compatibles. The driver is not normally accessed directly. Instead application code will use the API provided by the generic flash driver package `CYGPKG_IO_FLASH`, for example by calling functions like `cyg_flash_program`.

DataFlash devices are accessed via the SPI bus. Therefore, any platform on which this driver is to be used must also have an SPI driver. The DataFlash driver accesses the driver via the standard SPI API calls defined in the `CYGPKG_IO_SPI` package.

Configuration

The DataFlash flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package `CYGPKG_IO_FLASH` is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the DataFlash driver package.

Instantiating a DataFlash Device

Name

Instantiating — including the driver in an eCos target

Synopsis

```
#include <cyg/io/dataflash.h>

CYG_DATAFLASH_FLASH_DRIVER(char *name, cyg_spi_device *sdev, cyg_uint32 addr,
cyg_uint32 start, cyg_uint32 end);
```

Description

The DataFlash family contains several different flash devices, all supporting the same basic set of operations. The devices vary in capacity, performance and sector layout. There are also platform-specific issues such as which SPI bus the device is connected to, and which chip select it uses. The DataFlash driver package cannot know all this information. Instead it is the responsibility of another package, either the platform HAL or a flash configuration package, to instantiate some DataFlash device structures.

The definition of the parameters is split between two data structures. The first is an SPI specific data structure describing the SPI bus, chip select and signalling characteristics for the device. This is usually defined in an SPI specific configuration package or the platform HAL. The reader is referred to the SPI documentation for details of the contents of this structure, but see later for an example.

The second data structure defines the characteristics of the DataFlash device for use by the flash subsystem. For convenience a macro, `CYG_DATAFLASH_FLASH_DRIVER`, has been defined to automate the generation of this structure. This macro takes a number of arguments:

name

This provides a name fragment for distinguishing this DataFlash device from any others. It is concatenated with `cyg_dataflash_priv_` to form a static variable name. Any unique string that obeys the rules of C variable names is sufficient.

sdev

A pointer to the SPI device object that describes this flash device. If the SPI device has been declared with the name `spi_dataflash_dev0` then this argument should be `(&spi_dataflash_dev0.spi_device)`.

addr

SPI DataFlash devices do not have a physical address in the system memory space. However, the flash subsystem expects all flash devices to have an address. This argument gives the DataFlash device a virtual address in the memory space. This should be allocated to a location that contains no other flash devices, and to avoid

confusion, no other memory or devices. Subsequently, the DataFlash may be accessed by performing flash system accesses starting at this address.

start

This parameter defines the sector at which the flash device mapping starts. The beginning of this sector is mapped to the virtual address given in the `addr` argument. This value will usually be zero, but may be non-zero if there is reserved data at the beginning of the DataFlash.

end

This parameter defines the sector at which the flash device mapping ends. The end of this sector defines the maximum extent in the address space that the DataFlash occupies. The exact size of the mapping will depend on the number and sizes of the flash sectors covered. This value will usually be the number of sectors in the device, but may be less if there is reserved data at the end of the DataFlash.

Example

DataFlash support is usually specific to each platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the SPI bus, chip select and location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code is generally placed in the platform HAL or in a separate platform specific package.

The code to declare a DataFlash device might appear as follows:

```
#include <cyg/io/spi.h>
#include <cyg/io/spi_at91.h>
#include <cyg/io/dataflash.h>

__externC cyg_spi_at91_device_t spi_dataflash_dev0;

CYG_DATAFLASH_FLASH_DRIVER( eb55_dataflash,
                             (&spi_dataflash_dev0.spi_device),
                             0x08000000,
                             0,
                             16 );
```

Here, we are defining a dataflash device that is mapped to virtual address 0x08000000. The start and end sectors cover the entire flash device, 17 sectors. In addition to a DataFlash specific structure, this macro also creates a `cyg_flash_dev` structure which supplies the driver interface to the flash subsystem. The SPI device structure is defined elsewhere, in an SPI specific package, and has the following format:

```
#include <cyg/infra/cyg_type.h>
#include <cyg/io/spi.h>
#include <cyg/io/spi_at91.h>

// AT45DB321B DataFlash
cyg_spi_at91_device_t spi_dataflash_dev0 CYG_SPI_DEVICE_ON_BUS(0) =
{
    .spi_device.spi_bus = &cyg_spi_at91_bus.spi_bus,

    .dev_num      = 0,          // Device number
```

```

.cl_pol      = 1,          // Clock polarity (0 or 1)
.cl_pha      = 0,          // Clock phase (0 or 1)
.cl_brat     = 8192000,    // Clock baud rate
.cs_up_udly  = 1,          // Delay in usec between CS up and transfer start
.cs_dw_udly  = 1,          // Delay in usec between transfer end and CS down
.tr_bt_udly  = 1           // Delay in usec between two transfers
};

```

The parameters here attach the device to the only SPI bus in the hardware, use chip select 0 to access it, and set the communication parameters for the clock polarity, phase, baud rate and delays.

Device Info

The exact DataFlash device attached to the SPI bus is discovered by the driver by querying it and matching the device ID against an internal table of supported devices. If a particular device is not currently supported, it must be added to the table in `devs_flash_atmel_dataflash.c`. A typical entry in this table appears as follows:

```

{    // AT45DB321B
    device_id:    0x0D,
    page_size:    528,
    page_count:   8192,
    baddr_bits:   10,
    block_size:   8,
    sector_sizes: { 1, 63, 64, 64, 64, 64, 64, 64,
                    64, 64, 64, 64, 64, 64, 64, 64 },
    sector_count: 17
},

```

The fields of this structure are:

`device_id`

This defines the device ID returned as part of the status register. This is the field that is matched to select this DataFlash device.

`page_size`

This gives the size of pages in this flash device.

`page_size`

This gives the total number of pages in this device.

`baddr_bits`

This gives the number of bits used in the SPI command address format for specifying a byte address within a page.

`block_size`

This gives the number of pages in a block.

Instantiating a DataFlash Device

`sector_sizes`

This is an array giving the size, in blocks, of each sector of the DataFlash.

`sector_count`

This gives the number of entries in the *sector_sizes* array.

II. SST 39VFXXX Flash Device Driver

Overview

Name

Overview — eCos Support for SST 39VFXXX Flash Devices and Compatibles

Description

The `CYGPKG_DEVS_FLASH_SST_39VFXXX_V2` SST 39VFXXX V2 flash driver package implements support for the 39VFXXX family of flash devices and compatibles. Normally the driver is not accessed directly. Instead application code will use the API provided by the generic flash driver package `CYGPKG_IO_FLASH`, for example by calling functions like `cyg_flash_program`.

The driver imposes one restriction on application code which developers should be aware of: when programming the flash the destination addresses must be aligned to a bus boundary. For example if the target hardware has a single flash device attached to a 16-bit bus then program operations must involve a multiple of 16-bit values aligned to a 16-bit boundary. Note that it is the bus width that matters, not the device width. If the target hardware has two 16-bit devices attached to a 32-bit bus then program operations must still be aligned to a 32-bit boundary, even though in theory a 16-bit boundary would suffice. In practice this is rarely an issue, and requiring the larger boundary greatly simplifies the code and improves performance.

Note: Many eCos targets with 39vfxxx or compatible flash devices will still use the older driver package `CYGPKG_DEVS_FLASH_SST_39VFXXX`. Only newer ports and some older ports that have been converted will use the V2 driver. This documentation only applies to the V2 driver.

Configuration Options

The 39vfxxx flash driver package will be loaded automatically when configuring eCos for a target with suitable hardware. However the driver will be inactive unless the generic flash package `CYGPKG_IO_FLASH` is loaded. It may be necessary to add this generic package to the configuration explicitly before the driver functionality becomes available. There should never be any need to load or unload the AM29xxx driver package.

The driver contains a small number of configuration options which application developers may wish to tweak. `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_PROGRAM_BURST_SIZE` controls the program operation. On typical hardware programming the flash requires disabling interrupts and the cache for an extended period of time. Some or all of the flash hardware will be unusable while each word is programmed, and disabling interrupts is the only reliable way of ensuring that no interrupt handler or other thread will try to access the flash in the middle of an operation. This can have a major impact on the real-time responsiveness of the typical applications. To ameliorate this the driver will perform writes in small bursts, briefly re-enabling the cache and interrupts between each burst. The number of words written per burst is controlled by this configuration operation: reducing the value will improve real-time response but will add overhead, so the actual flash program operation will take longer; conversely more writes per burst will worsen response times but reduce overhead.

Similarly erasing a block of flash safely requires disabling interrupts and the cache. Erasing a block can easily take a second or so, and disabling interrupts for such a long period of time is usually undesirable. Hence the driver can also perform the erase in bursts, using the hardware's suspend and resume capabilities.

`CYGNUM_DEVS_FLASH_STRATA_V2_ERASE_BURST_DURATION` controls the number of polling loops during which interrupts are disabled. Reducing its value improves responsiveness at the cost of performance, and increasing its value has the opposite effect. Note that too low a value may prevent the erase operation from working at all: the chip will be spending its time suspending and resuming, rather than actually performing the erase. The minimum value will depend on the specific hardware.

There are a number of other options, relating mostly to hardware characteristics. It is very rare that application developers need to change any of these. For example the option `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_ERASE_REGIONS` may need a non-default value if the flash devices used on the target have an unusual boot block layout. If so the platform HAL will impose a requires constraint on this option and the configuration system will resolve the constraint. The only time it might be necessary to change the value manually is if the actual board being used is a variant of the one supported by the platform HAL and uses a different flash chip.

Instantiating an 39vfxxx Device

Name

Instantiating — including the driver in an eCos target

Synopsis

```
#include <cyg/io/39vfxxx_dev.h>

int cyg_39vfxxx_init_check_devid_XX(struct cyg_flash_dev* device);
int cyg_39vfxxx_init_cfi_XX(struct cyg_flash_dev* device);
int cyg_39vfxxx_erase_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr);
int cyg_39vfxxx_program_XX(struct cyg_flash_dev* device, cyg_flashaddr_t addr, const
void* data, size_t len);
int cyg_at49xxxx_softlock(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_at49xxxx_hardlock(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_at49xxxx_unlock(struct cyg_flash_dev* device, const cyg_flashaddr_t addr);
int cyg_39vfxxx_read_devid_XX(struct cyg_flash_dev* device);
```

Description

The 39VFXXX family contains several different flash devices, all supporting the same basic set of operations but with various common or uncommon extensions. The devices vary in capacity, performance, boot block layout, and width. There are also platform-specific issues such as how many devices are actually present on the board and where they are mapped in the address space. The 39vfxxx driver package cannot know the details of every chip and every platform. Instead it is the responsibility of another package, usually the platform HAL, to supply the necessary information by instantiating some data structures. Two pieces of information are especially important: the bus configuration and the boot block layout.

Flash devices are typically 8-bits, 16-bits, or 32-bits wide (64-bit devices are not yet in common use). Most 16-bit devices will also support 8-bit accesses, but not all. Similarly 32-bit devices can be accessed 16-bits at a time or 8-bits at a time. A board will have one or more of these devices on the bus. For example there may be a single 16-bit device on a 16-bit bus, or two 16-bit devices on a 32-bit bus. The processor's bus logic determines which combinations are possible, and there will be a trade off between cost and performance: two 16-bit devices in parallel can provide twice the memory bandwidth of a single device. The driver supports the following combinations:

8

A single 8-bit flash device on an 8-bit bus.

16

A single 16-bit flash device on a 16-bit bus.

32

A single 32-bit flash device on an 32-bit bus.

88

Two parallel 8-bit devices on an 16-bit bus.

8888

Four parallel 8-bit devices on a 32-bit bus.

1616

Two parallel 16-bit devices on a 32-bit bus, with one device providing the bottom two bytes of each 32-bit datum and the other device providing the top two bytes.

16as8

A single 16-bit flash device connected to an 8-bit bus.

These configuration all require slightly different code to manipulate the hardware. The 39vfxxx driver package provides separate functions for each configuration, for example `cyg_39vfxxx_erase_16` and `cyg_39vfxxx_program_1616`.

Caution

At the time of writing not all the configurations have been tested.

The second piece of information is the boot block layout. Flash devices are subdivided into blocks (also known as sectors - both terms are in common use). Some operations such as erase work on a whole block at a time, and for most applications a block is the smallest unit that gets updated. A typical block size is 64K. It is inefficient to use an entire 64K block for small bits of configuration data and similar information, so many flash devices also support a number of smaller boot blocks. A typical 2MB flash device could have a single 16K block, followed by two 8K blocks, then a 32K block, and finally 31 full-size 64K blocks. The boot blocks may appear at the bottom or the top of the device. So-called uniform devices do not have boot blocks, just full-size ones. The driver needs to know the boot block layout. With modern devices it can work this out at run-time, but often it is better to provide the information statically.

Example

In most cases flash support is specific to a platform. Even if two platforms happen to use the same flash device there are likely to be differences such as the location in the address map. Hence there is little possibility of re-using the platform-specific code, and this code should be placed in the platform HAL rather than in a separate package. Typically this involves a separate file and a corresponding compile property in the platform HAL's CDL:

```
cdl_package CYGPKG_HAL_M68K_ALAIA {  
    ...  
    compile -library=libextras.a alaia_flash.c  
    ...  
}
```

The contents of this file will not be accessed directly, only indirectly via the generic flash API, so normally it would be removed by link-time garbage collection. To avoid this the object file has to go into `libextras.a`.

The actual file `alaia_flash.c` will look something like:

```
#include <pkgconf/system.h>
#ifdef CYGPKG_DEVS_FLASH_SST_39VFXXX_V2

#include <cyg/io/flash.h>
#include <cyg/io/flash_dev.h>
#include <cyg/io/39vfxxx_dev.h>

static const CYG_FLASH_FUNCS(hal_alaia_flash_amd_funs,
    &cyg_39vfxxx_init_check_devid_16,
    &cyg_flash_devfn_query_nop,
    &cyg_39vfxxx_erase_16,
    &cyg_39vfxxx_program_16,
    (int (*)(struct cyg_flash_dev*, const cyg_flashaddr_t, void*, size_t))0,
    &cyg_flash_devfn_lock_nop,
    &cyg_flash_devfn_unlock_nop);

static const cyg_39vfxxx_dev hal_alaia_flash_priv = {
    .devid      = 0x45,
    .block_info = {
        { 0x00004000, 1 },
        { 0x00002000, 2 },
        { 0x00008000, 1 },
        { 0x00010000, 63 }
    }
};

CYG_FLASH_DRIVER(hal_alaia_flash,
    &hal_alaia_flash_amd_funs,
    0,
    0xFFC00000,
    0xFFFFFFFF,
    4,
    hal_alaia_flash_priv.block_info,
    &hal_alaia_flash_priv
);
#endif
```

The bulk of the file is protected by an `#ifdef` for the 39vfxxx flash driver. That driver will only be active if the generic flash support is enabled. Without that support there will be no way of accessing the device so instantiating the data structures would serve no purpose. The rest of the file is split into three structure definitions. The first supplies the functions which will be used to perform the actual flash accesses, using a macro provided by the generic flash code in `cyg/io/flash_dev.h`. The relevant ones have an `_16` suffix, indicating that on this board there is a single 16-bit flash device on a 16-bit bus. The second provides information specific to 39vfxxx flash devices. The third provides the `cyg_flash_dev` structure needed by the generic flash code, which contains pointers to the previous two.

Functions

All eCos flash device drivers must implement a standard interface, defined by the generic flash code `CYGPKG_IO_FLASH`. This interface includes a table of seven function pointers for various operations: initialization, query, erase, program, read, locking and unlocking. The query operation is optional and the generic flash support provides a dummy implementation `cyg_flash_devfn_query_nop`. 39vfxxx flash devices are always directly accessible so there is no need for a separate read function. The remaining functions are more complicated.

Usually the table can be declared `const`. In a ROM startup application this avoids both ROM and RAM copies of the table, saving a small amount of memory. `const` should not be used if the table may be modified by a platform-specific initialization routine.

Initialization

There is a choice of three main initialization functions. The simplest is `cyg_flash_devfn_init_nop`, which does nothing. It can be used if the `cyg_39vfxxx_dev` and `cyg_flash_dev` structures are fully initialized statically and the flash will just work without special effort. This is useful if it is guaranteed that the board will always be manufactured using the same flash chip, since the `nop` function involves the smallest code size and run-time overheads.

The next step up is `cyg_39vfxxx_init_check_devid_xx`, where `xx` will be replaced by the suffix appropriate for the bus configuration. It is still necessary to provide all the device information statically, including the `devid` field in the `cyg_39vfxxx_dev` structure. This initialization function will attempt to query the flash device and check that the provided device id matches the actual hardware. If there is a mismatch the device will be marked uninitialized and subsequent attempts to manipulate the flash will fail.

If the board may end up being manufactured with any of a number of different flash chips then the driver can perform run-time initialization, using a `cyg_39vfxxx_init_cfi_xx` function. This queries the flash device as per the Common Flash Memory Interface Specification, supported by all current devices (although not necessarily by older devices). The `block_info` field in the `cyg_39vfxxx_dev` structure and the `end` and `num_block_infos` fields in the `cyg_flash_dev` structure will be filled in. It is still necessary to supply the `start` field statically since otherwise the driver will not know how to access the flash device. The main disadvantage of using CFI is that it increases the code size.

Caution

If CFI is used then the `cyg_39vfxxx_dev` structure must not be declared `const`. The CFI code will attempt to update the structure and will fail if the structure is held in read-only memory. This would leave the flash driver non-functional.

A final option is to use a platform-specific initialization function. This may be useful if the board may be manufactured with one of a small number of different flash devices and the platform HAL needs to adapt to this. The 39vfxxx driver provides a utility function to read the device id, `cyg_39vfxxx_read_devid_xx`:

```
static int
alaia_flash_init(struct cyg_flash_dev* dev)
{
    int devid = cyg_39vfxxx_read_devid_1616(dev);
    switch(devid) {
        case 0x0042 :
            ...
    }
```



```

        case 0x0084 :
            ...
        default:
            return CYG_FLASH_ERR_DRV_WRONG_PART;
    }
}

```

There are many other possible uses for a platform-specific initialization function. For example initial prototype boards might have only supported 8-bit access to a 16-bit flash device rather than 16-bit access, but this problem was fixed in the next revision. The platform-specific initialization function can figure out which model board it is running on and replace the default 16as8 functions with faster 16 ones.

Erase and Program

The 39vfxxx driver provides erase and program functions appropriate for the various bus configurations. On most targets these can be used directly. On some targets it may be necessary to do some extra work before and after the erase and program operations. For example if the hardware has an MMU then the part of the address map containing the flash may have been set to read-only, in an attempt to catch spurious memory accesses. Erasing or programming the flash requires write-access, so the MMU settings have to be changed temporarily. As another example some flash device may require a higher voltage to be applied during an erase or program operation. or a higher voltage may be desirable to make the operation proceed faster. A typical platform-specific erase function would look like this:

```

static int
alaia_flash_erase(struct cyg_flash_dev* dev, cyg_flashaddr_t addr)
{
    int result;
    ... // Set up the hardware for an erase
    result = cyg_39vfxxx_erase_32(dev, addr);
    ... // Revert the hardware change
    return result;
}

```

There are two configurations which affect the erase and program functions, and which a platform HAL may wish to change: `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_ERASE_TIMEOUT` and `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_PROGRAM_TIMEOUT`. The erase and program operations both involve polling for completion, and these timeout impose an upper bound on the polling loop. Normally these operations should never take anywhere close to the timeout period, so a timeout indicates a catastrophic failure that should really be handled by a watchdog reset. A reset is particularly appropriate because there will be no clean way of aborting the flash operation. The main reason for the timeouts is to help with debugging when porting to new hardware. If there is a valid reason why a particular platform needs different timeouts then the platform HAL's CDL can require appropriate values for these options.

Locking

There is no single way of implementing the block lock and unlock operations on all 39vfxxx devices. If these operations are supported at all then usually they involve manipulating the voltages on certain pins. This would

not be able to be handled by generic driver code since it requires knowing how these pins can be manipulated via the processor's GPIO lines. Therefore the 39vfxxx driver does not usually provide lock and unlock functions, and instead the generic dummy functions `cyg_flash_devfn_lock_nop` and `cyg_flash_devfn_unlock_nop` should be used. An [exception](#) exists for the AT49xxxx family of devices which are sufficiently SST compatible in other respects. Otherwise, if a platform does provide a way of implementing the locking then this can be handled by platform-specific functions.

```
static int
alaia_lock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    ...
}

static int
alaia_unlock(struct cyg_flash_dev* dev, const cyg_flashaddr_t addr)
{
    ...
}
```

If real locking functions are implemented then the platform HAL's CDL script should implement the CDL interface `CYGHWR_IO_FLASH_BLOCK_LOCKING`. Otherwise the generic flash package may believe that none of the flash drivers in the system provide locking functionality and disable the interface functions.

AT49xxxx locking

As locking is standardised on across the AT49xxxx family of SST 39vfxxx compatible Flash parts, a method is supporting this is included within this driver. `cyg_at49xxxx_softlock_XX` provides a means of locking a Flash sector such that it may be subsequently unlocked. `cyg_at49xxxx_hardlock_XX` locks a sector such that it cannot be unlocked until after reset or a power cycle. `cyg_at49xxxx_unlock_XX` unlocks a sector that has previously been softlocked. At power on or Flash device reset, all sectors default to being softlocked.

Other

The driver provides a set of functions `cyg_39vfxxx_read_devid_XX`, one per supported bus configuration. These functions take a single argument, a pointer to the `cyg_flash_dev` structure, and return the chip's device id. For older devices this id is a single byte. For more recent devices the id is a 3-byte value, 0x7E followed by a further two bytes that actually identify the device. `cyg_39vfxxx_read_devid_XX` is usually called only from inside a platform-specific driver initialization routine, allowing the platform HAL to adapt to the actual device present on the board.

Device-Specific Structure

The `cyg_39vfxxx_dev` structure provides information specific to 39vfxxx flash devices, as opposed to the more generic flash information which goes into the `cyg_flash_dev` structure. There are only two fields: *devid* and *block_info*.

devId is only needed if the driver's initialization function is set to `cyg_39vfxxx_init_check_devid_XX`. That function will extract the actual device info from the flash chip and compare it with the *devId* field. If there is a mismatch then subsequent operations on the device will fail.

The *block_info* field consists of one or more pairs of the block size in bytes and the number of blocks of that size. The order must match the actual hardware device since the flash code will use the table to determine the start and end locations of each block. The table can be initialized in one of three ways:

1. If the driver initialization function is set to `cyg_flash_devfn_init_nop` or `cyg_39vfxxx_init_check_devid_XX` then the block information should be provided statically. This is appropriate if the board will also be manufactured using the same flash chip.
2. If `cyg_39vfxxx_init_cfi_XX` is used then this will fill in the block info table. Hence there is no need for static initialization.
3. If a platform-specific initialization function is used then either this should fill in the block info table, or the info should be provided statically.

The size of the *block_info* table is determined by the configuration option `CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_ERASE_REGIONS`. This has a default value of 4, which should suffice for nearly all 39vfxxx flash devices. If more entries are needed then the platform HAL's CDL script should require a larger value.

If the `cyg_39vfxxx_dev` structure is statically initialized then it can be `const`. This saves a small amount of memory in ROM startup applications. If the structure is updated at run-time, either by `cyg_39vfxxx_init_cfi_XX` or by a platform-specific initialization routine, then it cannot be `const`.

Flash Structure

Internally the generic flash code works in terms of `cyg_flash_dev` structures, and the platform HAL should define one of these. The structure should be placed in the `cyg_flashdev` table. The following fields need to be provided:

funs

This should point at the table of functions.

start

The base address of the flash in the address map. On some board the flash may be mapped into memory several times, for example it may appear in both cached and uncached parts of the address space. The *start* field should correspond to the cached address.

end

The address of the last byte in the flash. It can either be statically initialized, or `cyg_39vfxxx_init_cfi_XX` will calculate its value at run-time.

num_block_infos

This should be the number of entries in the *block_info* table. It can either be statically initialized or it will be filled in by `cyg_39vfxxx_init_cfi_XX`.

block_info

The table with the block information is held in the `cyg_39vfxxx_dev` structure, so this field should just point into that structure.

priv

This field is reserved for use by the device driver. For the 39vfxxx driver it should point at the appropriate `cyg_39vfxxx_dev` structure.

The `cyg_flash_dev` structure contains a number of other fields which are manipulated only by the generic flash code. Some of these fields will be updated at run-time so the structure cannot be declared `const`.

Multiple Devices

A board may have several flash devices in parallel, for example two 16-bit devices on a 32-bit bus. It may also have several such banks to increase the total amount of flash. If each device provides 2MB, there could be one bank of 2 parallel flash devices at 0xFF800000 and another bank at 0xFFC00000, giving a total of 8MB. This setup can be described in several ways. One approach is to define two `cyg_flash_dev` structures. The table of function pointers can usually be shared, as can the `cyg_39vfxxx_dev` structure. Another approach is to define a single `cyg_flash_dev` structure but with a larger *block_info* table, covering the blocks in both banks of devices. The second approach makes more efficient use of memory.

Many variations are possible, for example a small slow flash device may be used for initial bootstrap and holding the configuration data, while there is also a much larger and faster device to hold a file system. Such variations are usually best described by separate `cyg_flash_dev` structures.

If more than one `cyg_flash_dev` structure is instantiated then the platform HAL's CDL script should implement the CDL interface `CYGHWR_IO_FLASH_DEVICE` once for every device past the first. Otherwise the generic code may default to the case of a single flash device and optimize for that.

Platform-Specific Macros

The 39vfxxx driver source code includes the header files `cyg/hal/hal_arch.h` and `cyg/hal/hal_io.h`, and hence indirectly the corresponding platform header files (if defined). Optionally these headers can define macros which are used inside the driver, thus giving the HAL limited control over how the driver works.

Cache Management

By default the 39vfxxx driver assumes that the flash can be accessed uncached, and it will use the HAL `CYGARC_UNCACHED_ADDRESS` macro to map the cached address in the *start* field of the `cyg_flash_dev` structure into an uncached address. If for any reason this HAL macro is inappropriate for the flash then an alternative macro `HAL_39VFXXX_UNCACHED_ADDRESS` can be defined instead. However fixing the `CYGARC_UNCACHED_ADDRESS` macro is normally the better solution.

If there is no way of bypassing the cache then the platform HAL should implement the CDL interface `CYGHWR_DEVS_FLASH_SST_39VFXXX_V2_CACHED_ONLY`. The flash driver will now disable and re-enable the cache as required. For example a program operation will involve the following:

```

AM29_INTSCACHE_STATE;
AM29_INTSCACHE_BEGIN();
while ( ! finished ) {
    write a burst of CYGNUM_DEVS_FLASH_SST_39VFXXX_V2_PROGRAM_BURST_SIZE
    AM29_INTSCACHE_SUSPEND();
    AM29_INTSCACHE_RESUME();
}
AM29_INTSCACHE_END();

```

The default implementations of these INTSCACHE macros are as follows: `STATE` defines any local variables that may be needed, e.g. to save the current interrupt state; `BEGIN` disables interrupts, synchronizes the data caches, disables it, and invalidates the current contents; `SUSPEND` re-enables the data cache and then interrupts; `RESUME` disables interrupts and the data cache; `END` re-enables the cache and then interrupts. The cache is only disabled when interrupts are disabled, so there is no possibility of an interrupt handler running or a context switch occurring while the cache is disabled, potentially leaving the system running very slowly. The data cache synchronization ensures that there are no dirty cache lines, so when the cache is disabled the low-level flash write code will not see stale data in memory. The invalidate ensures that at the end of the operation higher-level code will not pick up stale cache contents instead of the newly written flash data. The `SUSPEND` and `RESUME` macros only re-enable and disable the data cache. An interrupt and possibly a context switch may occur between these macros and use the cache normally. It is assumed that any code which runs at this time will not touch the memory being used by the flash operation, so as far as the low-level program code is concerned it can just continue to use the uncached memory contents as set up by the `BEGIN` macro. If any code modifies the const data currently being written to a flash block or tries to read the flash block being modified then the system's behaviour is undefined. Theoretically a more robust approach is possible, synchronizing and invalidating the cache again in every `RESUME`. However these cache operations can be expensive and `RESUME` may get invoked some thousands of times for every flash block, so this alternative approach would cripple the driver's performance.

Some implementations of the HAL cache macros may not provide the exact semantics required by the flash driver. For example `HAL_DCACHE_DISABLE` may have an unwanted side effect, or it may do more work than is needed here. The driver will check for alternative macros `HAL_39VFXXX_INTSCACHE_STATE`, `HAL_39VFXXX_INTSCACHE_BEGIN`, `HAL_39VFXXX_INTSCACHE_SUSPEND`, `HAL_39VFXXX_INTSCACHE_RESUME` and `HAL_39VFXXX_INTSCACHE_END`, using these instead of the defaults.

III. SMSC LAN9118 Ethernet Driver

Instantiating an 39vfxxx Device

SMSC LAN9118 Ethernet Driver

Name

CYGPKG_DEVS_ETH_SMSC_LAN9118 — eCos Support for SMSC LAN9118 Ethernet Devices

Description

The SMSC LAN9118 chip is a high performance single chip ethernet controller which can be interfaced to a variety of embedded processors. This package provides an eCos driver for that device. The driver supports both polled mode for use by RedBoot and interrupt-driven mode for use by a full TCP/IP stack.

The exact interface between the LAN9118 chip and the main processor is determined by the platform HAL. On some platforms there may even be multiple LAN9118 chips. This package only provides the platform-independent code. It is up to the platform HAL to instantiate one or more device instances and to provide information such as the base address and interrupt vector. There is also no explicit support for features like auto-negotiation or advanced flow control. These are left to the platform HAL or to the application, although usually the default settings will be acceptable for most applications.

Configuration Options

This package should be loaded automatically when selecting a target equipped with a LAN9118 ethernet chip, and it should never be necessary to load it explicitly. If the application does not actually require ethernet functionality then the package is inactive and the final executable will not suffer any overheads from unused functionality. This is determined by the presence of the generic ethernet I/O package CYGPKG_IO_ETH_DRIVERS. Typically the choice of eCos template causes the right thing to happen. For example the default template does not include any TCP/IP stack so CYGPKG_IO_ETH_DRIVERS is not included, but both the net and redboot templates do include a TCP/IP stack so will specify that package and hence enable the ethernet driver.

Optionally the ethernet driver can maintain statistics about the number of incoming and transmitted ethernet frames, receive overruns, collisions, and other conditions. Maintaining and providing these statistics involves some overhead, and is controlled by the configuration option CYGFUN_DEVS_ETH_SMSC_LAN9118_STATISTICS. Typically these statistics are only accessed through SNMP, so by default statistics gathering is enabled if the configuration includes CYGPKG_SNMPAGENT and disabled otherwise.

Porting the Driver to New Hardware

It is the responsibility of the platform HAL to instantiate one or more devices, depending on the number of LAN9118 chips present. Typically this involves a separate file in the platform HAL sources:

```
#include <cyg/io/lan9118.h>

LAN9118_INSTANCE(alaia, 0, "eth0", alaia_eth_init);

static bool
alaia_eth_init(struct cyg_netdevtab_entry* tab)
{
```

```

...

return cyg_lan9118_eth_init(tab);
}

```

The first two arguments to the LAN9118_INSTANCE macro identify the platform and the device instance, and are used to construct unique variable names. The third argument gives the device name, and the final argument is a platform-specific initialization function. The platform HAL should also contain suitable CDL to build this file:

```

cdl_component CYGHWR_HAL_ALAIA_ETH {
display  "External ethernet support"
parent   CYGPKG_IO_ETH_DRIVERS
flavor   none
active_if CYGPKG_IO_ETH_DRIVERS
implements CYGNUM_DEVS_ETH_SMSC_LAN9118_COUNT
compile  -library=libextras.a alaia_eth.c
description "
    The Alaia board comes with a single LAN9118 ethernet device."

cdl_option CYGNUM_HAL_ALAIA_ETH_ISR_PRIORITY {
    ...
}
...
}

```

If the configuration does not include the generic ethernet support then this component will be inactive. Otherwise the file `alaia_eth.c` will get built into `libextras.a`, ensuring the device instance does not get eliminated by linker garbage collection. The interface `CYGNUM_DEVS_ETH_SMSC_LAN9118_COUNT` should be implemented once per LAN9118 chip. If additional configuration options are needed, for example to control the MAC address or the interrupt priority, then these can go inside the component.

The driver needs to know where to access the device. If there is a single LAN9118 chip then the required information can be supplied via `#define`'s in the `plf_io.h` header:

```

#define HAL_LAN9118_BASE          0xBA000000
#define HAL_LAN9118_ISRVEC        CYGNUM_HAL_ISR_LAN9118
#define HAL_LAN9118_ISRPRI        CYGNUM_HAL_ALAIA_ETH_ISR_PRIORITY

```

Otherwise the platform-specific initialization function should put this information in fields in the LAN9118 instance structure:

```

static bool
alaia_eth_init(struct cyg_netdevtab_entry* tab)
{
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_base   = 0xBA000000;
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_isrvec = CYGNUM_HAL_ISR_PIO4;
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_isrpri = 1;
    LAN9118_INSTANCE_NAME(alaia, 1).lan9118_base   = 0xBB000000;
    LAN9118_INSTANCE_NAME(alaia, 1).lan9118_isrvec = CYGNUM_HAL_ISR_PIO5;
    LAN9118_INSTANCE_NAME(alaia, 1).lan9118_isrpri = 1;
}

```

```

...
return cyg_lan9118_eth_init(tab);
}

```

The initialization function should ensure that the processor's bus interface is set up correctly for talking to the ethernet chip, and that the interrupt vector has been configured correctly for level vs. edge interrupts. This must happen before calling the driver init function. Also the *lan9118_hw_flags* should be set correctly as per the flags in *lan9118.h*, for example:

```

static bool
alaia_eth_init(struct cyg_netdevtab_entry* tab)
{
    ...
    LAN9118_INSTANCE_NAME(alaia, 0).lan9118_hw_flags =
        (LAN9118_HW_FLAGS_IRQ_POL_ACTIVE_HIGH      |
         LAN9118_HW_FLAGS_IRQ_PUSH_PULL           |
         LAN9118_HW_FLAGS_HAS_LED1                 |
         LAN9118_HW_FLAGS_HAS_LED2);
    ...
    return cyg_lan9118_eth_init(tab);
}

```

The LAN9118 can be accessed via either a 16-bit or 32-bit bus, and from big-endian or little-endian processors. This gives a number of combinations. The chip is inherently little-endian, so on a little-endian processor there should be no problems. On a big-endian processor there are two possibilities. If the LAN9118 is interfaced in the obvious way then it will be necessary to swap the data of all incoming and outgoing packets, which imposes a significant performance penalty. On a 16-bit bus the *LAN9118_HW_FLAGS_16BIT_BE* flag should be set. Alternatively the bytes on the bus can be swapped, either by the hardware or by programming the processor's bus interface. This means no swapping is needed for data, but all accesses to the LAN9118's command and status registers need swapping instead. However most of that swapping can be done at compile-time so has no overhead. Defining *HAL_LAN9118_SWAP_COMMANDS* in *plf_io.h* sets up this mode. If there are multiple ethernet chips then the driver assumes they are all wired the same way. For further details consult the driver's source code.

All ethernet devices require a unique MAC address. Ideally this will be provided by a serial EEPROM or similar, and if such a device is present and attached to the LAN9118 then it will be used automatically by the ethernet chip to set the MAC address. If the platform does not have a suitable EEPROM then the MAC address must come from elsewhere, for example a RedBoot fconfig option, and the platform-specific initialization function should fill in the instance's *lan9118_mac* field.

IV. ST M48T35A Wallclock Device Driver

ST M48T35A Wallclock Device Driver

Name

CYGPKG_DEVS_WALLCLOCK_ST_M48T35A — eCos Support for the ST M48T35A Timekeeper SRAM chips and compatibles

Description

This package CYGPKG_DEVS_WALLCLOCK_ST_M48T35A provides a device driver for the wallclock device in the ST M48T35AY and M48T35AV timekeeper SRAM chips. These combine 32K of battery-backed SRAM and a real-time clock in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support CYGPKG_IO_WALLCLOCK. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as `time` and `gmtime`. The eCos C library also provides a non-standard function `cyg_libc_time_settime` for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

M48T35A devices provide some support for a calibration value. If the application has some alternative way of getting a reliable time value, for example NTP over a TCP/IP network, then the wallclock can be made to tick slightly faster or slower. The calibration value is a small integer between -31 and +31. A positive value x adds $512x$ extra cycles every 125829120 actual cycles, speeding up the clock by approximately $10.7x$ seconds per month. Alternatively a negative value x subtracts $256x$ cycles, slowing down the clock by $5.35x$ seconds per month. The package provides two functions for examining and changing the current calibration value:

```
#include <cyg/io/wallclock_m48t35a.h>

externC cyg_int32  cyg_wallclock_m48t35a_get_calibration(void);
externC void       cyg_wallclock_m48t35a_set_calibration(cyg_int32);
```

Porting

For most platforms adding support for the m48t35a wallclock device requires just two steps. The package must be added to the appropriate CDL target entry so that it gets loaded automatically whenever configuring eCos for

that target. Also the platform HAL should specify the location of the clock hardware in the address space, by defining the symbol `HAL_WALLCLOCK_M48T35A_BASE`. The definition should go into `cyg/hal/hal_io.h` or more commonly into a platform-specific header `cyg/hal/plf_io.h` which gets included automatically by the former. The value should be the address of the control register of the clock device. For example, given a battery-backed 32K timekeeper chip at 0x30000000, the clock hardware will occupy the last eight bytes at 0x30007ff8 and that is the value which should be used.

The package provides some support for hardware where the clock is mapped into memory in strange ways. The platform HAL can define an additional symbol `HAL_WALLCLOCK_M48T35A_STRIDE` and macros `HAL_WALLCLOCK_M48T35A_READ_UINT8` and `HAL_WALLCLOCK_M48T35A_WRITE_UINT8` to change the way in which the driver accesses the hardware. The source code should be consulted for further details of how these work.

Some variants of the M48T35A do not support the century bit. If the platform defines `HAL_WALLCLOCK_M48T35A_NO_CENTURY_BIT` then the century bit will be ignored and the driver will instead use a heuristic for determining the century: if the year register < 70 then this is treated as relative to 2000; otherwise it is treated as relative to 1900; this gives an effective range of Jan 1st 1970 to Dec 31st 2069.

V. lwIP - the lightweight IP stack for eCosPro[®]

Chapter 1. lwIP overview

Introduction

lwIP, short for lightweight IP, is an implementation of a standard Internet Protocol v4 networking protocol stack designed to operate in a resource-constrained environment. It was created in 2001 by Adam Dunkels (<http://www.sics.se/~adam/lwip/>) of the Swedish Institute of Computer Science (<http://www.sics.se/>) for his Master's thesis (<http://www.sics.se/~adam/publications.html#theses>). The core lwIP code was released publicly under an open licence.

The lwIP stack supports the IP, TCP, UDP, ICMP, ARP, DHCP, SLIP and PPP protocols, and there is a selection of APIs which applications can use to interact with it. As well as being designed from the outset to have a low memory footprint, it also gains many of its lightweight properties from being highly configurable. This makes it an excellent choice for integration into eCos.

This documentation describes lwIP and properties specific to its port to eCosPro®. The usage, configuration and tuning of lwIP will also be discussed. Many of the concepts discussed here will require some understanding of the inherent underlying properties of the TCP, UDP and IP protocols. This documentation cannot substitute for an introduction to TCP/IP stacks and protocols generally, and it is recommended that where needed the reader seeks out a good reference book, such as:

- *TCP/IP Illustrated, Volume 1: The Protocols*, W. Richard Stevens, published by Addison-Wesley Professional, ISBN-10: 0-201-63346-9, ISBN-13: 978-0-201-63346-7.
- *Internetworking with TCP/IP: volume 1*, Douglas E. Comer, published by Prentice-Hall, ISBN-10: 0-131-87671-6, ISBN-13: 978-0-131-87671-2.

or one of the many online guides:

- The TCP/IP Guide (<http://www.tcpipguide.com/>)
- Network Sorcery RFC Sourcebook (<http://www.networksorcery.com/enp/default.htm>)
- Wikipedia (http://en.wikipedia.org/wiki/Internet_protocol_suite)

lwIP sources and ports

lwIP is portable and by no means specific to eCos. It has an active development community and undergoes continuous development of its core code, focussed around its project page (<http://savannah.nongnu.org/projects/lwip/>) on the Savannah (<http://savannah.nongnu.org/>) development site run by the Free Software Foundation (<http://www.fsf.org/>).

The lwIP project releases do include a port to eCos, but that is one of the very first versions of the port, is obsolete and is to be removed.

The public eCos project includes a slightly updated version of the port. However, there are many problems in the port which will result in failure in operation, slower operation, unscalability, and a much larger resource footprint. In addition, the configuration controls are inaccurate and incomplete.

As a result, eCosCentric® decided that in order to provide a robust, feature-rich, and commercially supportable solution for eCosPro®, the port had to be completely overhauled, and mostly rewritten.

This documentation corresponds solely to the eCosPro® port of lwIP, and the usage, configuration system and operation differs in many regards from that in other code bases. In addition, there are several fixes and feature improvements to the core lwIP code base, some of which are exclusive to eCosPro®.

External documentation

A limited amount of publically available documentation is available for the lwIP project. Some of it has been incorporated into this manual. The following lists useful documentation known about at the time of writing:

- Adam Dunkel's Master's Thesis (<http://www.sics.se/~adam/publications.html#theses>) - the original description of lwIP design and operation, but now somewhat outdated.
- Report by Adam Dunkel into the design and implementation of lwIP (<http://www.sics.se/~adam/lwip/documentation.html>), including a sequential API reference, and example code. Largely still applicable to current lwIP, albeit incomplete. Available in PDF (<http://www.sics.se/~adam/lwip/doc/lwip.pdf>) and compressed postscript (.ps.gz) (<http://www.sics.se/~adam/lwip/doc/lwip.ps.gz>). A copy of the PDF version may be found in the `doc/` subdirectory of the lwIP package in the eCosPro source repository (`packages/net/lwip_tcpip/VERSION/doc/dunkels-lwip.pdf` relative to the base of the eCosPro installation).
- Text description of the lwIP raw API (<http://cvs.savannah.gnu.org/viewcvs/lwip/lwip/doc/rawapi.txt?rev=1&view=auto>). A copy of the version at time of writing may be found in the `doc/` subdirectory of the lwIP package in the eCosPro source repository (`packages/net/lwip_tcpip/VERSION/doc/rawapi.txt` relative to the base of the eCosPro installation).
- Text description of the sys_arch porting abstraction layer (http://cvs.savannah.gnu.org/viewcvs/lwip/lwip/doc/sys_arch.txt?rev=1&view=auto). A copy of the version at time of writing may be found in the `doc/` subdirectory of the lwIP package in the eCosPro source repository (`packages/net/lwip_tcpip/VERSION/doc/sys_arch.txt` relative to the base of the eCosPro installation).

Licensing

The lwIP core code is distributed under a 3 clause BSD-style license (http://www.fsf.org/licensing/licenses/index_html#ModifiedBSD). Confirmation has been received from Adam Dunkels that the existing public lwIP documentation is also covered by this license.

The original public eCos port included elements distributed under the eCos license (<http://ecos.sourceforge.org/license-overview.html>).

As a result of the substantial changes made by eCosCentric, significant portions of the eCos port of lwIP in eCosPro® are covered by the eCosPro License (<http://www.ecoscentric.com/ecospro-license.shtml>).

Chapter 2. Basic concepts

Structure

lwIP has been incorporated into eCos as a single package (`CYGPKG_NET_LWIP`) which contains all the core lwIP code and the bulk of the eCos port. The remaining elements that constitute the eCos port can be found in the generic Ethernet driver package (`CYGPKG_IO_ETH_DRIVERS`) and so is only relevant when using an Ethernet-based network card rather than SLIP or PPP. SLIP/PPP support is layered over the standard eCos serial driver API.

The port to eCos has been constructed using the `sys_arch` porting abstraction within lwIP, and this allows the eCos port to be cleanly separated from the core lwIP code, although it still remains in the lwIP eCos package.

Application Programming Interfaces (APIs)

There are three different APIs which may be used by applications to interface with the stack: the raw API, the sequential API, or the BSD sockets compatibility API. Each one in turn builds on the functionality provided by the previous API. This allows users the flexibility of choosing a fairly bare implementation to squeeze the maximum out of the available resources; or to use a more powerful API to simplify application coding and reduce time-to-market. Note that despite the presence of the BSD sockets compatibility API, the lwIP stack implementation is not in any way related to the other BSD-derived TCP/IP stacks present in eCos.

The raw API provides an event-based interface with callbacks directly into the application in order to handle incoming/outgoing data and events. There is no inter-thread protection and can only operate with a single thread of execution.

The sequential API is a more traditional style of network interface API which provides functions that may be called synchronously to perform network operations, and where those operations can be considered complete (or will complete asynchronously with no further application interaction) when those functions return to the application.

When using the sequential API (or the BSD sockets API which is layered on top of it), lwIP maintains its own internal thread for network data processing and event management. This is usually referred to as lwIP's TCP/IP thread (even though that is a slight misnomer). This thread uses mailboxes to communicate with application threads, and semaphores to provide mutual exclusion protection.

The BSD sockets compatibility API included in lwIP provides a subset of the Berkeley sockets interface introduced in the BSD 4.2 operating system. The Berkeley sockets interface, recently standardised by ISO/IEC in POSIX 2003.1, will be familiar to those who have developed network applications on Linux, POSIX, UNIX or to a limited extent Windows with Winsock.

As the BSD sockets API provided as part of lwIP is only a subset of the full sockets, it should be considered only as an aid to development or for when porting existing code. It should not be considered as a drop-in replacement for applications written for a complete BSD network stack implementation which supports a wealth of features that do not exist in, and in many cases would be inappropriate for, a low footprint implementation such as lwIP.

Protocol implementations

lwIP implements a variety of protocols. Support for each protocol can be individually included in or excluded from the configuration, subject to dependency constraints. The protocol implementations are mostly compartmentalised into separate source modules. Support exists for TCP, UDP, IP obviously (and implicitly ICMP), ARP when using Ethernet, DHCP, SLIP and PPP.

In most cases functionality has been intentionally restricted to avoid "bloat" (unnecessary features increasing resource use), or in some cases completely omitted. This is covered in slightly more detail in [the Section called Limitations](#).

Packet data buffers

lwIP does not only possess features allowing it by itself to maintain a small footprint, but also has design aspects which allow it to work with the application to reduce footprint. One important case of this is lwIP packet data buffers.

Packet data buffers in lwIP are termed *pbufs*. Pbufs can be chained together in fairly arbitrary ways to create a *pbuf chain*. The idea is that the application can pass the stack a pbuf of data to transmit, and the stack can prepend and possibly append other pbufs to encapsulate the data in protocol headers/footers without having to copy the data elsewhere, thus saving resources. In some cases, depending on precisely how the pbuf was allocated, the stack may even be able to fit protocol headers inside the pbuf passed to it. It also means that the application can itself provide data allocated in differing ways and from different locations, but assembled together as a pbuf chain. This will ensure that the data is treated as if it were all allocated contiguously. When using the sequential API, the underlying pbufs are wrapped in a *netbuf* construct in order to provide a simpler API to manipulate data in buffers; but the underlying functionality remains based on pbufs.

When a pbuf is created, it must be one of a variety of types:

PBUF_RAM

This is a conventional buffer, which points to data allocated from a pool in RAM managed by lwIP. On creation the buffer size must be given.

PBUF_ROM

This is a buffer pointing to immutable read-only data. This allows fixed literal data to be stored in ROM/Flash rather than using up precious RAM. Note that data pointed at by a PBUF_ROM pbuf does not literally have to point at read-only memory. All it means is that the data must not change, even if control has returned to the application. The pbuf data may still be being referenced as part of a packet waiting in a queue to be transmitted, or more often, waiting in a queue in case retransmission is necessary.

PBUF_REF

This is a buffer pointing to mutable data, passed in by reference. This means data provided by the application allocated from its own resources, and which could change in the future. This differs from PBUF_RAM packets in that the data is allocated by the application, and not from lwip's PBUF_RAM buffer memory pool. As the application could change the data after control is returned to it, if lwIP finds it must enqueue the pbuf, it will internally copy the data to a new PBUF_RAM. The benefits of this type of packet occur when the packet does not need to be enqueued, and so no PBUF_RAM pbuf needs to be allocated.

Configurability

lwIP was designed from the outset to have a low resource footprint. One of the techniques it uses to achieve this goal is its high level of configurability.

lwIP allows both coarse- and fine-grained control of functionality. Large sections of potentially unused functionality can be selected to be removed by the user, including entire protocol stacks. Such examples of removable coarse-grained functionality include UDP, TCP, SLIP, PPP stacks, ethernet/ARP support, IP fragmentation and/or reassembly, or the sequential API.

It has a somewhat modular and layered design to assist with this. It is intentionally only somewhat modular: other TCP/IP stacks have strictly enforced interfaces and abstractions between protocol layers. These abstractions are frequently cumbersome and can result in unnecessary resource implications. lwIP deliberately violates some of these protocol interface layering abstractions where doing so could improve resource utilisation. An example is reserving an estimated appropriate amount of space for protocol headers when constructing packets, where the choice of protocol dictates the amount of space.

Where lwIP really stands out is in its fine-grained control over the various pools of resources. Most resources are compartmentalised into fixed size memory pools to allow sizes to be constrained deterministically. The application designer will know, or can choose, the maximum number of network connections which are to be supported depending on application requirements. They also know the level of data throughput required for transmission or reception and can control the levels of the necessary resources appropriately, such as numbers of buffers (separately for incoming or outgoing packet data) and their sizes, numbers of protocol control blocks, TCP window sizes and more.

In this way, application designers can choose a configuration that maximises performance within the limitations of available memory. Clearly, the more constrained the memory, the greater the potential for adverse consequences for performance, or the number of supported connections. However, it should be realised that even with copious quantities of memory resources available to lwIP, it cannot be expected that a stack intentionally designed from the outset to be sparing with memory will perform as well as a stack intentionally designed from the outset for high performance. Nevertheless careful tuning of lwIP almost always results in significant performance improvements.

Limitations

As already mentioned, lwIP does not seek to provide a complete implementation of a TCP/IP stack providing the same level of functionality provided in large OSes such as Linux, Windows, *BSD, etc. While some aspects are controlled by configuration, in other cases functionality is intentionally limited to fit the design requirements of a compact footprint.

While a complete list of the limitations would be too numerous to enumerate, here are some of the most relevant ones to be aware of:

- Retransmission and windowing algorithms are implemented simply, at the expense of some performance.
- Routing is simplified - one gateway per interface. IP forwarding follows the same rules as the host itself.
- No support for IGMP, NAT, nor packet filtering.
- The TCP, DHCP and IP protocols can contain options in their packets. Relatively few of these options are supported by lwIP.

- Path MTU discovery (from RFC1191 (<http://www.faqs.org/rfcs/rfc1191.html>)) is not supported. Ordinarily it is used to avoid fragmentation of packets resulting from the maximum MTU of an intermediate link between source and destination being smaller than the packet sizes actually transmitted. lwIP does however allow the TCP Maximum Segment Size (MSS) to be configured.
- No complex data structures, caches and search trees to optimise speed. Generally simple lists are used.
- Thread safety (for the sequential and BSD compatibility API) is implemented in a very simple form. Individual connections should not be operated on by multiple threads simultaneously. The mutual exclusion that is provided is at a very coarse grain - the network processing operations themselves are not multi-threaded.
- Selective Acknowledgements (SACKs) (from RFC2018 (<http://www.faqs.org/rfcs/rfc2018.html>)) are not provided in the TCP implementation. SACKs are a commonly implemented approach to increasing performance on links subject to packet loss, packet errors or congestion.
- Most ICMP packet types are ignored.
- If IP fragmentation and reassembly support is enabled, no more than one sequence of IP fragments received at one point on *any* connection can be reassembled at one time. Even for each connection, overlapping fragments from different packets are not handled. In such cases, the packet is simply dropped. It is hoped, that a retransmission may be successful. Received IP fragments are allowed to be reassembled out of order however.
- The BSD sockets compatibility API does not implement all socket options (including SO_REUSEADDR/SO_REUSEPORT), API functions, nor API semantics.
- Error handling for application errors is frequently only handled with asserts - used only during debug builds during development, allowing for smaller production code in release builds.
- The TCP persist timer is not implemented. If a remote peer has filled its receive window and as a result lwIP stops sending, then when the remote peer processes more data it sends an ACK to update the window. However if that ACK is lost, then if data is entirely unidirectional (lwIP to remote host), the connection could stall. In practice, this has not been something people have experienced really.
- TCP data is not split in the unsent queue, resulting in somewhat inefficient use of receiver windows.

There are many more examples.

Quick Start

Incorporating lwIP into your application is straightforward. The essential starting point is to incorporate the lwIP eCos package (CYGPKG_NET_LWIP) into your configuration.

This may be achieved directly using **ecosconfig add** on the command line, or the **Build->Packages...** menu item within the eCos Configuration Tool. If you wish to support Ethernet devices, you will also need to include the “Common Ethernet Support” (CYGPKG_IO_ETH_DRIVERS) eCos package. For SLIP/PPP support, you will need to enable the “Hardware serial device drivers” (CYGPKG_IO_SERIAL_DEVICES) configuration option within the “Serial device drivers” (CYGPKG_IO_SERIAL) eCos package.

Alternatively, as a convenience, configuration templates have been provided to permit an easy starting point for creating a configuration incorporating lwIP. Two templates are provided: `lwip_eth` for those intending to use lwIP with Ethernet; and `lwip_ppp` for those intending to use lwIP with PPP. These may be used either by providing the template name as an extra argument on the command line to **ecosconfig add**; or with the **Build->Templates...** menu item within the eCos Configuration Tool. Both these templates are basic, incorporating only those packages which are essential for lwIP operation.

At this stage it would be appropriate to tailor the lwIP package configuration to the application requirements. At a minimum it would be appropriate to consider whether a static IP address, or a dynamic IP address served from a DHCP server, is required. Note that if RedBoot is used on the target and incorporates network support, then you must not give lwIP and RedBoot the same IP address. For the same reason, you must not configure both lwIP and RedBoot to obtain an IP address via DHCP.

If obtaining an address via DHCP it can be convenient to enable the network interface debugging configuration option within lwIP (`CYGDBG_LWIP_DEBUG_NETIF`). This will allow the IP address which was set to be viewed on the diagnostic output console.

Prior to coding your application to perform lwIP stack operations using its APIs, the stack must be initialised. This does not happen automatically, and instead a C function must be called:

```
int cyg_lwip_init(void);
```

The function declaration can be obtained by including the `network.h` header file:

```
#include <network.h>
```

`cyg_lwip_init` returns 0 on success and non-zero on failure. Note that 0 may be returned even if no network interfaces were successfully initialised. This is because in some cases interfaces are brought up asynchronously in any case, devaluing such an error indication; and because an interface not coming up may be expected. If the application needs to determine the status of interfaces, it should query the stack using the `netif_*` functions using the `<lwip/netif.h>` header file.

The `cyg_lwip_init` function *must* be called from a thread context. Raw API users need not call this function, although they instead will be required to perform their own stack initialisation. Consult the [raw API documentation](#) for more information.

Chapter 3. Port

Port status

The eCos port of lwIP in eCosPro is based on the lwIP 1.1.1 code base. Due to a large number of identified code problems, performance and footprint inefficiencies, the eCos port was mostly rewritten. Modifications consisting of both bug fixes and feature enhancements were made to the lwIP core code.

The port requires the eCos kernel (`CYGPKG_KERNEL`) for now. The main reasons for this are because the ethernet driver and serial driver implementations have dependencies on interrupts and non-kernel interrupt support is tricky; and that it is only really feasible in the lwIP core code to avoid a multi-thread OS if solely using the raw API. And when using the raw API, the application would have to be responsible for polling the underlying device driver (e.g. Ethernet) in any case.

Some eCos Ethernet drivers may have alignment constraints on packet data. This is usually not a problem, however it can affect PBUF_ROM packets, whose alignment is dictated by the application. Therefore the application must ensure only appropriately aligned PBUF_ROM packets are passed to lwIP, as appropriate for the hardware-specific Ethernet device driver.

lwIP's BSD sockets compatibility API is completely separate from the socket and file descriptor interface provided by the eCos File I/O (`CYGPKG_IO_FILEIO`) package. As such, network packages which rely on semantics such as being able to read and write both files and sockets with that API, cannot work with lwIP at the present time. This includes the httpd, DNS, SNTP and FTP client packages. The NET-SNMP package uses BSD stack-specific APIs and so also cannot work with lwIP. Note that an example httpd server written using the lwIP raw API is included in the `tests/` subdirectory of the lwIP eCos package.

C++ users should note that lwIP's own headers are not C++ safe, and so inclusions of these header files should be surrounded by:

```
extern "C" {  
    #include <lwip/...>  
}
```

blocks. However for convenience when using the BSD sockets compatibility API, including the `network.h` header file:

```
#include <network.h>
```

allows access to the API and works from C++ as well as C. This also has the benefit of potentially allowing interchangeable application code if switching between the lwIP BSD socket compatibility API and the real BSD stack port in eCos.

lwIP does not attempt to provide a cleanly delineated namespace for lwIP functions. This could make it difficult to port legacy code where there is a chance of conflicting names and symbols, both functions and data. Care is required here.

The serial-based SLIP and PPP protocols should be functional, however they have not been well tested, and so are not supported under the terms of incident support in eCosPro.

There is only convenient configuration for a single SLIP and/or PPP interface. Multiple interface support is planned for some future point.

IPv6 is considered experimental and is at a very early stage, and so is not included. Similarly SNMP support was very recently added to the lwIP code base, but while being promising, it is considered experimental and so is not included.

Chapter 4. Sequential API

Overview

As described [earlier](#), the lwIP sequential API provides a straightforward and easy-to-use method of interfacing to the stack. Unlike the raw API, which requires event-driven callbacks, an application can simply call the API functions as needed to perform stack operations such as sending data, receiving data, or manipulating packet buffers or connections. While the raw API may allow for more efficient operation, the sequential API typically allows for simpler application design.

Comparison with BSD sockets

In design, it is not unlike the BSD sockets API. Some of the terminology differs however: in the sequential API, the term *connection* is used for any communication link between network peers, and the handle for a connection is termed a *netconn*. A netconn can be considered analogous to a socket, albeit specific to networking - BSD sockets traditionally represent both network connections and files.

The main reason for superiority over the socket API occurs with buffer management. The BSD socket API was designed to manage the fact that the user and the operating system kernel operate in different address spaces and data must always be copied regardless. This results in not only decreased performance, but also increased footprint as buffers must be allocated to hold the copied data.

Netbufs

Instead the sequential API uses *netbufs*, which are based on [pbufs](#). This allows users to manage buffers directly, including even allowing data to come from ROM. Since pbufs, and hence netbufs, can be chained, this also allows the application and lwIP to avoid the need for large regions of entirely contiguous memory in order to hold data. Instead data can be constructed in chunks, and chained together.

When the application wishes to send data, it can send a netbuf directly with UDP. TCP is different as it is intrinsically a buffering, streaming protocol, which requires data to be kept aside to allow for retransmissions. As a result data is sent using just a pointer to memory and a length. However since TCP data can also reside in ROM, it is possible to indicate that the data does not need copying, and so will persist even if the stack needs to queue the data. This can lead to huge savings of memory. For example, static web page content can reside in ROM, and never need to be copied to RAM.

For both TCP and UDP, incoming data is passed to the application as netbufs. The application can use API functions to extract the data from the netbufs - care must be taken as the received data may in fact be a chain. A convenience function exists to copy out the entirety of data across the whole chain into a single contiguous region of memory. Otherwise the application can process data in each netbuf in the chain in turn. The functions `netbuf_first()` and `netbuf_next()` can be used to iterate through the chain.

TCP/IP thread

When interacting with the network stack using the sequential API, all operations are not handled by the calling thread, but instead are passed to the lwIP network processing (TCP/IP) thread. Inter-thread communication is used inside lwIP to ensure that at the point the API function returns, operation is either complete, or for asynchronous operations, under way.

Usage

API declarations

Declarations for all sequential API types and functions may be obtained by including the `<lwip/api.h>` header file:

```
#include <lwip/api.h>
```

Types

Objects of type `struct netconn` and `struct netbuf` are intended to be used as opaque types and the structure contents are intended to be maintained and viewed only by lwIP itself. User applications accessing internal members do so at their own risk, and future API compatibility is not guaranteed, nor is thread synchronisation since lwIP is entitled to change structure contents at any time.

IP address representation

Some API functions take an argument of type `struct ip_addr`. The type may be accessed as if it has the following structure:

```
struct ip_addr {
    u32_t addr;
};
```

Caution

API users must use the declaration of this structure from the header file `<lwip/ip_addr.h>` which is included implicitly by `<lwip/api.h>`. This type must not be declared by the application itself.

For convenience, predefined `struct ip_addr` instances are provided for the special cases of "any" IP address (0.0.0.0), and the global broadcast address (255.255.255.255). These instances can be accessed with the macro defines `IP_ADDR_ANY` and `IP_ADDR_BROADCAST` which return values of type `struct ip_addr *`.

The *addr* field is a 32-bit integral value representing the IP address in network byte order (not host byte order).

A variety of convenience function-like macros exist for manipulation or evaluation of IP addresses:

`IP_ADDR_ANY`

This macro evaluates to an expression of type `struct ip_addr *` identifying an IP address structure which can be used to represent the special "any" IP address 0.0.0.0.

`IP_ADDR_BROADCAST`

This macro evaluates to an expression of type `struct ip_addr *` identifying an IP address structure which can be used to represent the special global IP address 255.255.255.255.

`IN_CLASSA(a)`

An expression which evaluates to non-zero if *a* (of type `u32_t` and in host byte order) is a class A internet address.

`IN_CLASSB(a)`

An expression which evaluates to non-zero if *a* (of type `u32_t` and in host byte order) is a class B internet address.

`IN_CLASSC(a)`

An expression which evaluates to non-zero if *a* (of type `u32_t` and in host byte order) is a class C internet address.

`IN_CLASSD(a)`

An expression which evaluates to non-zero if *a* (of type `u32_t` and in host byte order) is a class D internet address.

`IP4_ADDR(ipaddr, a, b, c, d)`

Sets *ipaddr* (of type `struct ip_addr *`) to the internet address a.b.c.d. For example:

```
struct ip_addr host;
IP4_ADDR(&host, 192, 168, 1, 1);
```

`ip_addr_cmp(addr1, addr2)`

Returns non-zero if the arguments *addr1* and *addr2*, both of type `struct ip_addr *` are identical. Zero if they differ.

`ip_addr_netcmp(addr1, addr2, mask)`

Returns non-zero if the arguments *addr1* and *addr2*, both of type `struct ip_addr *` are on the same network, as indicated by the network mask *mask* which is itself also of type `struct ip_addr *`. Zero if they are on different networks.

`htons(s)`

Portably converts *s* of type `u16_t` from host byte order to a `u16_t` in network byte order.

`ntohs(s)`

Portably converts *s* of type `u16_t` from network byte order to a `u16_t` in host byte order.

`htonl(l)`

Portably converts *l* of type `u32_t` from host byte order to a `u32_t` in network byte order.

`ntohl(l)`

Portably converts *l* of type `u32_t` from network byte order to a `u32_t` in host byte order.

Some further potentially useful macro definitions can be viewed in `<lwip/ip_addr.h>`.

Error codes

While the BSD sockets API uses POSIX standard error codes (`ENOMEM`, `EINVAL`, etc.) the lwIP sequential API has its own separate set of error code definitions.

These error definitions are used by any API function that returns a value of type `err_t`. The following table indicates possible error code values and their meaning:

Table 4-1. lwIP sequential API error codes

Code	Meaning
<code>ERR_OK</code>	No error, operation successful.
<code>ERR_MEM</code>	Out of memory error.
<code>ERR_BUF</code>	Buffer error.
<code>ERR_ABRT</code>	Connection aborted.
<code>ERR_RST</code>	Connection reset.
<code>ERR_CLSD</code>	Connection closed.
<code>ERR_CONN</code>	Not connected.
<code>ERR_VAL</code>	Illegal value.
<code>ERR_ARG</code>	Illegal argument.
<code>ERR_RTE</code>	Routing problem.
<code>ERR_USE</code>	Address in use.
<code>ERR_IF</code>	Low-level network interface error.
<code>ERR_ISCONN</code>	Already connected.

API reference

netbuf_new()

Name

`netbuf_new()` — Allocate a netbuf structure

Synopsis

```
struct netbuf *netbuf_new(void);
```

Description

Allocates a netbuf structure. No buffer space is allocated when doing this, only the top level structure. After use, the netbuf must be deallocated with `netbuf_delete()`.

netbuf_new()

netbuf_delete()

Name

`netbuf_delete()` — Deallocate a netbuf structure

Synopsis

```
void netbuf_delete(struct netbuf *);
```

Description

Deallocates a netbuf structure previously allocated by a call to the `netbuf_new()` function. Any buffer memory allocated to the netbuf by calls to `netbuf_alloc()` is also deallocated.

Example

Example 4-1. This example shows the basic mechanisms for using netbufs.

```
int
main()
{
    struct netbuf *buf;
    buf = netbuf_new();      /* create a new netbuf */
    netbuf_alloc(buf, 100); /* allocate 100 bytes of buffer */

    /* do something with the netbuf */
    /* [...] */
    netbuf_delete(buf); /* deallocate netbuf */
}
```

netbuf_delete()

netbuf_alloc()

Name

`netbuf_alloc()` — Allocate space in a netbuf

Synopsis

```
void *netbuf_alloc(struct netbuf *buf, u16_t size);
```

Description

Allocates buffer memory with *size* number of bytes for the netbuf *buf*. The function returns a pointer to the allocated memory. Any memory previously allocated to the netbuf *buf* is deallocated. The allocated memory can later be deallocated with the [netbuf_free\(\)](#) function. Since protocol headers are expected to precede the data when it should be sent, the function allocates memory for protocol headers as well as for the actual data.

netbuf_alloc()

netbuf_free()

Name

`netbuf_free()` — Deallocate buffer memory associated with a netbuf

Synopsis

```
void netbuf_free(struct netbuf *buf);
```

Description

Deallocates the buffer memory associated with the netbuf *buf*. If no buffer memory has been allocated for the netbuf, this function does nothing.

netbuf_free()

netbuf_ref()

Name

`netbuf_ref()` — Associate a data pointer with a netbuf

Synopsis

```
void netbuf_ref(struct netbuf *buf, void *data, ul6_t size);
```

Description

Associates the external memory pointed to by the *data* pointer with the netbuf *buf*. The size of the external memory is given by *size*. Any memory previously allocated to the netbuf is deallocated. The difference between allocating memory for the netbuf with `netbuf_alloc()` and allocating memory using, e.g., `malloc()` and referencing it with `netbuf_ref()` is that in the former case, space for protocol headers is allocated as well which makes processing and sending the buffer faster.

Example

Example 4-1. This example shows a simple use of the `netbuf_ref()`

```
int
main()
{
    struct netbuf *buf;
    char string[] = "A string";

    /* create a new netbuf */
    buf = netbuf_new();

    /* reference the string */
    netbuf_ref(buf, string, sizeof(string));

    /* do something with the netbuf */
    /* [...] */

    /* deallocate netbuf */
    netbuf_delete(buf);
}
```

netbuf_ref()

netbuf_len()

Name

`netbuf_len()` — Obtain the total length of a netbuf

Synopsis

```
u16_t netbuf_len(struct netbuf *buf);
```

Description

Returns the total length of the data in the netbuf *buf*, even if the netbuf is fragmented. For a fragmented netbuf, the value obtained by calling this function is not the same as the size of the first fragment in the netbuf.

netbuf_len()

netbuf_data()

Name

`netbuf_data()` — Obtain a pointer to netbuf data

Synopsis

```
err_t netbuf_data(struct netbuf *buf, void **data, ul6_t *len);
```

Description

This function is used to obtain a pointer to and the length of a block of data in the netbuf *buf*. The arguments *data* and *len* are result parameters that will be filled with a pointer to the data and the length of the data pointed to. If the netbuf is fragmented, this function gives a pointer to one of the fragments in the netbuf. The application program must use the fragment handling functions `netbuf_first()` and `netbuf_next()` in order to reach all data in the netbuf. See the example under `netbuf_next()` for an example of how use `netbuf_data()`.

netbuf_data()

netbuf_next()

Name

`netbuf_next()` — Traverse internal fragments in a netbuf

Synopsis

```
s8_t netbuf_next(struct netbuf *buf);
```

Description

This function updates the internal fragment pointer in the netbuf *buf* so that it points to the next fragment in the netbuf. The return value is zero if there are more fragments in the netbuf, > 0 if the fragment pointer now points to the last fragment in the netbuf, and < 0 if the fragment pointer already pointed to the last fragment.

Example

Example 4-1. This example shows how to use the `netbuf_next()` function

We assume that this is in the middle of a function and that the variable *buf* is a netbuf.

```
/* [...] */
do {
    char *data;
    int len;

    /* obtain a pointer to the data in the fragment */
    netbuf_data(buf, &data, &len);

    /* do something with the data */
    do_something(data, len);
} while(netbuf_next(buf) >= 0);
/* [...] */
```

netbuf_next()

netbuf_first()

Name

`netbuf_first()` — Reset fragment pointer to start of netbuf

Synopsis

```
void netbuf_first(struct netbuf *buf);
```

Description

Resets the internal fragment pointer in the netbuf *buf* so that it points to the first fragment.

netbuf_first()

netbuf_copy()

Name

`netbuf_copy()` — Copy all netbuf data to memory pointer

Synopsis

```
void netbuf_copy(struct netbuf *buf, void *data, ul6_t len);
```

Description

Copies all of the data from the netbuf *buf* into the memory pointed to by *data* even if the netbuf *buf* is fragmented. The *len* parameter is an upper bound of how much data that will be copied into the memory pointed to by *data*.

Example

Example 4-1. This example shows a simple use of `netbuf_copy()`

Here, 200 bytes of memory is allocated on the stack to hold data. Even if the netbuf *buf* has more data than 200 bytes, only 200 bytes are copied into *data*.

```
void
example_function(struct netbuf *buf)
{
    char data[200];
    netbuf_copy(buf, data, 200);

    /* do something with the data */
}
```

netbuf_copy()

netbuf_copy_partial()

Name

`netbuf_copy_partial()` — Copy some netbuf data to memory pointer

Synopsis

```
void netbuf_copy_partial(struct netbuf *buf, void *data, ul6_t len, ul6_t offset);
```

Description

This function is similar to [netbuf_copy\(\)](#) except that it takes an extra parameter, *offset*, which can be used to set an offset from the start of the packet to start copying the *len* bytes.

netbuf_copy_partial()

netbuf_chain()

Name

`netbuf_chain()` — Chain two netbufs together

Synopsis

```
void netbuf_chain(struct netbuf *head, struct netbuf *tail);
```

Description

Chains the two netbufs *head* and *tail* together so that the data in *tail* will become the last fragment(s) in *head*. The netbuf *tail* is deallocated and should not be used after the call to this function.

netbuf_chain()

netbuf_fromaddr()

Name

`netbuf_fromaddr()` — Obtain the sender's IP address for a netbuf

Synopsis

```
struct ip_addr *netbuf_fromaddr(struct netbuf *buf);
```

Description

Returns the IP address of the host the netbuf *buf* was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function `netbuf_fromport()` can be used to obtain the port number of the remote host.

netbuf_fromaddr()

netbuf_fromport()

Name

`netbuf_fromport()` — Obtain the sender's port number for a netbuf

Synopsis

```
u16_t netbuf_fromport(struct netbuf *buf);
```

Description

Returns the port number of the host the netbuf *buf* was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function [netbuf_fromaddr\(\)](#) can be used to obtain the IP address of the remote host.

netbuf_fromport()

netconn_new()

Name

`netconn_new()` — Create a new connection structure

Synopsis

```
struct netconn *netconn_new(enum netconn_type type);
```

Description

Creates a new connection abstraction structure. The argument usually one of either `NETCONN_TCP` or `NETCONN_UDP`, yielding either a TCP or a UDP connection. No connection is established by the call to this function and no data is sent over the network.

For more advanced use, it is also possible to specify different connection types: `NETCONN_UDPLITE`, `NETCONN_UDPNOCHKSUM` or `NETCONN_RAW`.

netconn_new()

netconn_new_with_callback()

Name

`netconn_new_with_callback()` — Create a new connection structure with a callback

Synopsis

```
struct netconn *netconn_new_with_callback(enum netconn_type type, void (*callback)
(struct netconn *conn, enum netconn_evt evt, ul6_t len));
```

Description

This function is similar to `netconn_new()` except that an additional function pointer `callback` is passed. The function pointed to by `callback` will be called when data is sent or received. Specifically, the `netconn_evt` parameter to the callback is used to indicate the event type. This enum can have the following values:

NETCONN_EVT_RCVPLUS

Used when new incoming data from a remote peer arrives. The amount of data received is passed in `len`. If `len` is 0 then a connection event has occurred: this may be an error, the acceptance of a connection for a listening connection (called for the listening connection), or deletion of the connection.

NETCONN_EVT_RCVMINUS

Used when new incoming data from a remote peer has been received and accepted by higher layers. The amount of data accepted is passed in `len`. If `len` is 0 then this indicates the acceptance of a connection as a result of a listening port (called for the newly created accepted connection).

NETCONN_EVT_SENDPLUS

Used when data has been sent to a remote peer and received by it. This only occurs for TCP connections, and specifically is only triggered when, as a consequence of TCP acknowledgements from the remote peer, the free TCP send buffer size now exceeds the configured send buffer low water mark (configured with the `CYGNUM_LWIP_TCP_SNDLOWAT` CDL configuration option). The amount of data sent in the most recent transaction is passed in `len`. If `len` is 0 then this indicates the connection has been deleted.

NETCONN_EVT_SENDMINUS

This is only used for TCP connections, and is triggered when a sufficient amount of data has been sent on the connection that the amount of free send buffer space is now under the send buffer low water mark (configured with the `CYGNUM_LWIP_TCP_SNDLOWAT` CDL configuration option). The amount of data sent in the most recent transaction is passed in `len`.

netconn_new_with_callback()

netconn_new_with_proto_and_callback()

Name

`netconn_new_with_proto_and_callback()` — Create a new connection structure with a callback for a specific protocol

Synopsis

```
struct netconn *netconn_new_with_callback(enum netconn_type type, u16_t proto, void (*callback) (struct netconn *conn, enum netconn_evt evt, u16_t len));
```

Description

This function is similar to [netconn_new_with_callback\(\)](#) except that an additional parameter *proto* may be used to indicate the IP protocol number to use. If *proto* is non-zero, it must only be used with the *type* set to `NETCONN_RAW`.

The most common use of this function is the creation of connections suitable for generating ICMP packets.

netconn_new_with_proto_and_callback()

netconn_delete()

Name

`netconn_delete()` — Deallocate a netconn

Synopsis

```
err_t netconn_delete(struct netconn *conn);
```

Description

Deallocates the netconn *conn*. If the connection is open, it is closed as a result of this call.

netconn_delete()

netconn_type()

Name

`netconn_type()` — Obtain the type of netconn

Synopsis

```
enum netconn_type netconn_type(struct netconn *conn);
```

Description

Returns the type of the connection *conn*. This is the same type that is given as an argument to `netconn_new()` (and its variants) and can be one of `NETCONN_TCP`, `NETCONN_UDP`, `NETCONN_UDPLITE`, `NETCONN_UDPNOCHKSUM` or `NETCONN_RAW`.

netconn_type()

netconn_peer()

Name

`netconn_peer()` — Obtain the remote host IP address/port of a netconn

Synopsis

```
err_t netconn_peer(struct netconn *conn, struct ip_addr *addr, ul6_t *port);
```

Description

This function is used to obtain the IP address and port of the remote end of the connection indicated by *conn*. The parameters *addr* and *port* are result parameters that are set by the function. If the connection *conn* is not connected to any remote host, the results are undefined.

netconn_peer()

netconn_addr()

Name

`netconn_addr()` — Obtain the local host IP address/port of a netconn

Synopsis

```
err_t netconn_addr(struct netconn *conn, struct ip_addr **addr, ul6_t *port);
```

Description

This function is used to obtain the local IP address and port number of the connection *conn*.

netconn_addr()

netconn_bind()

Name

`netconn_bind()` — Set local IP address/port of a netconn

Synopsis

```
err_t netconn_bind(struct netconn *conn, struct ip_addr *addr, u16_t port);
```

Description

Binds the connection *conn* to the local IP address *addr* and TCP or UDP port *port*. If *addr* is `NULL`, the local IP address is determined by the networking system.

netconn_bind()

netconn_connect()

Name

`netconn_connect()` — Connect netconn to remote peer

Synopsis

```
err_t netconn_connect(struct netconn *conn, struct ip_addr *remote_addr, u16_t  
remote_port);
```

Description

In case of UDP, sets the remote receiver as given by *remote_addr* and *remote_port* of UDP messages sent over the connection. For TCP, `netconn_connect()` opens a connection with the remote host.

Solely for UDP, it is possible to call `netconn_connect()` repeatedly to set a new remote destination to use for UDP packets, rather than having to create and delete netconns for each destination.

netconn_connect()

netconn_disconnect()

Name

`netconn_disconnect()` — Disconnect UDP connection

Synopsis

```
err_t netconn_disconnect(struct netconn *conn);
```

Description

This function is only relevant for UDP connections. It unsets any previously set (using [netconn_connect\(\)](#)) remote peer address and port associated with connection *conn*.

netconn_disconnect()

netconn_listen()

Name

`netconn_listen()` — Make a listening TCP netconn

Synopsis

```
err_t netconn_listen(struct netconn *conn);
```

Description

Puts the TCP connection *conn* into the TCP LISTEN state. This means its purpose will become listening for incoming connections from remote peers. `netconn_accept()` is required to establish a connection resulting from incoming connection requests.

netconn_listen()

netconn_accept()

Name

`netconn_accept()` — Wait for incoming connections

Synopsis

```
struct netconn *netconn_accept(struct netconn *conn);
```

Description

This function blocks the process until a connection request from a remote host arrives on the TCP connection *conn*. The connection must be in the LISTEN state so `netconn_listen()` must be called prior to `netconn_accept()`. When a connection is established with the remote host, a new connection structure is returned.

Example

Example 4-1. This example shows how to open a TCP server on port 2000

Note: This is only an example for illustrative purposes, and a complete version should perform comprehensive error checking.

```
int
main()
{
    struct netconn *conn, *newconn;

    /* create a connection structure */
    conn = netconn_new(NETCONN_TCP);

    /* bind the connection to port 2000 on any local IP address */
    netconn_bind(conn, NULL, 2000);

    /* tell the connection to listen for incoming connection requests */
    netconn_listen(conn);

    /* block until we get an incoming connection */
    newconn = netconn_accept(conn);
```

```
netconn_accept()

    /* do something with the connection */
    process_connection(newconn);

    /* deallocate both connections */
    netconn_delete(newconn);
    netconn_delete(conn);
}
```

netconn_recv()

Name

`netconn_recv()` — Wait for data

Synopsis

```
struct netbuf *netconn_recv(struct netconn *conn);
```

Description

This function blocks the process while waiting for data to arrive on the connection *conn*. If the connection has been closed by the remote host, NULL is returned, otherwise a netbuf containing the received data is returned.

Example

Example 4-1. This example demonstrates usage of the `netconn_recv()` function

In the following code, we assume that a connection has been established before the call to `example_function()`.

```
void
example_function(struct netconn *conn)
{
    struct netbuf *buf;

    /* receive data until the other host closes the connection */
    while((buf = netconn_recv(conn)) != NULL) {
        do_something(buf);
    }

    /* the connection has now been closed by the other end, so we close our end */
    netconn_close(conn);
}
```

netconn_recv()

netconn_write()

Name

`netconn_write()` — Send data on TCP connection

Synopsis

```
err_t netconn_write(struct netconn *conn, void *data, u16_t len, u8_t copy);
```

Description

This function is only used for TCP connections. It puts the data pointed to by *data* on the output queue for the TCP connection *conn*. The length of the data is given by *len*. There is no restriction on the length of the data. This function does not require the application to explicitly allocate buffers, as this is taken care of by the stack. The *copy* parameter has two possible states, as shown below:

```
#define NETCONN_NOCOPY 0x00  
#define NETCONN_COPY 0x01
```

When passed the flag `NETCONN_COPY` the data is copied into internal buffers which are allocated for the data. This allows the data to be modified directly after the call, but is inefficient both in terms of execution time and memory usage. If the flag `NETCONN_NOCOPY` is used, the data is not copied but rather referenced. The data must not be modified after the call, since the data can be put on the retransmission queue for the connection, and stay there for an indeterminate amount of time. This is useful when sending data that is located in ROM and therefore is immutable. If greater control over the modifiability of the data is needed, a combination of copied and non-copied data can be used, as seen in the example below.

Example

Example 4-1. This example demonstrates basic usage of the `netconn_write()` function

Here, the variable *data* is assumed to be modified later in the program, and is therefore copied into the internal buffers by passing the flag `NETCONN_COPY` to `netconn_write()`. The text variable contains a string that will not be modified and can therefore be sent using references instead of copying.

Note: This is only an example for illustrative purposes, and a complete version should perform comprehensive error checking.

netconn_write()

```
int
main()
{
    struct netconn *conn;
    char data[10];
    char text[] = "Static text";
    int i;

    /* set up the connection conn */
    /* [...] */

    /* create some arbitrary data */
    for(i = 0; i < 10; i++)
        data[i] = i;

    netconn_write(conn, data, 10, NETCONN_COPY);
    netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);

    /* the data can be modified */
    for(i = 0; i < 10; i++)
        data[i] = 10 - i;

    /* take down the connection conn */
    netconn_close(conn);
}
```


netconn_send()

Name

`netconn_send()` — Send data on UDP connection

Synopsis

```
err_t netconn_send(struct netconn *conn, struct netbuf *buf);
```

Description

Send the data in the netbuf *buf* on the UDP connection *conn*. The data in the netbuf should not be too large if IP fragmentation support is disabled. If IP fragmentation support is disabled, the data should not be larger than the maximum transmission unit (MTU) of the outgoing network interface, less the space required for link layer, IP and UDP headers. No checking is necessarily made of whether the data is sufficiently small and sending very large netbufs might give undefined results.

Example

Example 4-1. This example demonstrates basic usage of the `netconn_send()` function

This example shows how to send some UDP data to UDP port 7000 on a remote host with IP address 10.0.0.1.

Note: This is only an example for illustrative purposes, and a complete version should perform comprehensive error checking.

```
int
main()
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr addr;
    char *data;
    char text[] = "A static text";
    int i;

    /* create a new connection */
    conn = netconn_new(NETCONN_UDP);
```

netconn_send()

```
/* set up the IP address of the remote host */
addr.addr = htonl(0x0a000001);

/* connect the connection to the remote host */
netconn_connect(conn, &addr, 7000);

/* create a new netbuf */
buf = netbuf_new();
data = netbuf_alloc(buf, 10);

/* create some arbitrary data */
for(i = 0; i < 10; i++)
    data[i] = i;

/* send the arbitrary data */
netconn_send(conn, buf);

/* reference the text into the netbuf */
netbuf_ref(buf, text, sizeof(text));

/* send the text */
netconn_send(conn, buf);

/* deallocate connection and netbuf */
netconn_delete(conn);
netconn_delete(buf);
}
```

netconn_close()

Name

`netconn_close()` — Close a connection

Synopsis

```
err_t netconn_close(struct netconn *conn);
```

Description

Close the connection *conn*.

netconn_close()

netconn_err()

Name

`netconn_err()` — Obtain connection error status

Synopsis

```
err_t netconn_err(struct netconn *conn);
```

Description

Obtain the stored error status of connection *conn*.

netconn_err()

Chapter 5. Raw API

Much of the information in this chapter has been derived from lwIP's own raw API documentation (<http://cvs.savannah.gnu.org/viewcvs/lwip/lwip/doc/rawapi.txt?rev=1.7&view=log>), although additions, modifications and adaptations for eCos have been made.

Overview

While the high level [lwIP sequential API](#) is good for programs that are themselves sequential and can benefit from the blocking open-read-write-close paradigm, lwIP itself is event based by nature. If an application can be written with an event-based approach, then it becomes possible to integrate directly with the event-based design of the core lwIP code.

The *raw TCP/IP API* allows the application program to integrate better with the TCP/IP code. Program execution is event based by having callback functions being called from within the TCP/IP code. The TCP/IP code and the application program both run in the same thread. The sequential API has a much higher overhead and is not very well suited for small systems since it forces a multithreaded paradigm on the application.

The raw TCP/IP interface is not only faster in terms of code execution time but is also less memory intensive. The drawback is that program development is somewhat harder and application programs written for the raw TCP/IP interface are more difficult to understand. Still, this is the preferred way of writing applications that should be small in code size and memory usage.

Both APIs can be used simultaneously by different application programs. In fact, the sequential API is implemented as an application program using the raw TCP/IP interface.

An example of an application using the raw API can be found in the `tests/` subdirectory of the lwIP eCos package. This test is built when the CDL configuration option `CYGBLD_NET_LWIP_BUILD_MANUAL_TESTS` is enabled. This raw API application acts as a simple HTTP server.

Usage

The raw API is a very direct interface, and is "close to the metal". In particular, the mechanism by which incoming packets are injected into the TCP/IP stack, or outgoing packets are sent by the stack are largely left to the application. This means that it is generally better not to use the `cyg_lwip_init()` function that is used for sequential or BSD API applications. Instead the application should perform its own stack initialisation.

Note that if you do decide to use `cyg_lwip_init()` and the configuration option `CYGFUN_LWIP_SEQUENTIAL_API` is disabled so that solely the raw API is available, then the application will need to provide its own alternative to the `tcpip_input()` function which had previously been used to inject received packets into the stack. This function must be declared as follows:

```
err_t tcpip_input(struct pbuf *, struct netif *);
```

See below for further details on initialisation.

Declarations for the API functions are found in header files within the lwIP include tree. The TCP functions are found in `<lwip/tcp.h>`, and UDP in `<lwip/udp.h>`.

The raw API uses many of the same types and definitions used in the sequential API. In particular the raw API functions use [struct ip_addr](#) and [err_t error codes](#).

Callbacks

Program execution is driven by callbacks. Each callback is an ordinary C function that is called from within the TCP/IP code. Every callback function is passed the current TCP or UDP connection state as an argument. Also, in order to be able to keep program specific state, the callback functions are called with a program specified argument that is independent of the TCP/IP state.

The `tcp_arg()` function is used for setting the application connection state.

tcp_arg()

Name

`tcp_arg()` — Set the application connection state

Synopsis

```
void tcp_arg(struct tcp_pcb *pcb, void *arg);
```

Description

The `tcp_arg()` function specifies the program specific state that should be passed to all other callback functions. The "pcb" argument is the current TCP connection control block, and the "arg" argument is the argument that will be passed to the callbacks.

TCP connection setup

The functions used for setting up connections are similar to those of the sequential API and of the BSD socket API. A new TCP connection identifier (i.e., a protocol control block - PCB) is created with the `tcp_new()` function. This PCB can then be either set to listen for new incoming connections or be explicitly connected to another host.

tcp_new()

Name

`tcp_new()` — Create a new TCP PCB

Synopsis

```
struct tcp_pcb *tcp_new(void);
```

Description

Creates a new TCP connection identifier (PCB).

Return value

Returns the new PCB. If memory is not available for creating the new PCB, `NULL` is returned.

tcp_new()

tcp_bind()

Name

`tcp_bind()` — Bind PCB to local IP address and port

Synopsis

```
err_t tcp_bind(struct tcp_pcb *pcb, struct ip_addr *ipaddr, u16_t port);
```

Description

Binds *pcb* to a local IP address and port number. The IP address can be specified as `IP_ADDR_ANY` in order to bind the connection to all local IP addresses.

Return value

If another connection is bound to the same port, the function will return `ERR_USE`, otherwise `ERR_OK` is returned.

tcp_bind()

tcp_listen()

Name

`tcp_listen()` — Make PCB listen for incoming connections

Synopsis

```
struct tcp_pcb *tcp_listen(struct tcp_pcb *pcb);
```

Description

Commands *pcb* to start listening for incoming connections. When an incoming connection is accepted, the function specified with the `tcp_accept()` function will be called. *pcb* must have been bound to a local port with the `tcp_bind()` function.

Return value

The `tcp_listen()` function returns a new connection identifier, and the one passed as an argument to the function will be deallocated. The reason for this behavior is that less memory is needed for a connection that is listening, so `tcp_listen()` will reclaim the memory needed for the original connection and allocate a new smaller memory block for the listening connection.

`tcp_listen()` may return `NULL` if no memory was available for the listening connection. If so, the memory associated with *pcb* will not be deallocated.

tcp_listen()

tcp_accept()

Name

`tcp_accept()` — Set callback used for new incoming connections

Synopsis

```
void tcp_accept(struct tcp_pcb *pcb, err_t (*accept) (void *arg, struct tcp_pcb  
*newpcb, err_t err));
```

Description

Specify the callback function that should be called when a new connection arrives for a listening TCP PCB.

tcp_accept()

tcp_connect()

Name

`tcp_connect()` — Open connection to remote host

Synopsis

```
err_t tcp_connect(struct tcp_pcb *pcb, struct ip_addr *ipaddr, u16_t port, err_t
(*connected) (void *arg, struct tcp_pcb *tpcb, err_t err));
```

Description

Sets up *pcb* to connect to the remote host indicated by *ipaddr* on port *port* and sends the initial *SYN* segment which opens the connection.

The `tcp_connect()` function returns immediately; it does not wait for the connection to be properly set up. Instead, it will call the `connected()` function specified as the fourth argument when the connection is established. If the connection could not be properly established, either because the other host refused the connection or because the other host didn't answer, the `connected()` function will be called with its *err* argument set accordingly.

Return value

The `tcp_connect()` function can return `ERR_MEM` if no memory is available for enqueueing the *SYN* segment. If the *SYN* indeed was enqueued successfully, the `tcp_connect()` function returns `ERR_OK`.

tcp_connect()

Sending TCP data

TCP data is sent by enqueueing the data with a call to `tcp_write()`. When the data is successfully transmitted to the remote host, the application will be notified with a call to a specified callback function.

tcp_write()

Name

`tcp_write()` — Enqueue data for transmission

Synopsis

```
err_t tcp_write(struct tcp_pcb *pcb, const void *dataptr, u16_t len, u8_t copy);
```

Description

Enqueues the data pointed to by *dataptr*. The length of the data is passed in *len*. The argument *copy* may be either 0 or 1 and indicates whether the new memory should be allocated for the data to be copied into. If the argument is 0, no new memory should be allocated and the data should only be referenced by pointer.

Return value

The `tcp_write()` function will fail and return `ERR_MEM` if the length of the data exceeds the current send buffer size (as defined by the `CYGNUM_LWIP_TCP_SND_BUF` CDL configuration option) or if the length of the queue of outgoing segment is larger than the upper limit defined by the `CYGNUM_LWIP_TCP_SND_QUEUELEN` CDL configuration option. The number of bytes available in the output queue can be retrieved with the `tcp_sndbuf()` function:

```
u16_t tcp_sndbuf(struct tcp_pcb *pcb);
```

The proper way to use this function is to call the function with at most `tcp_sndbuf()` bytes of data. If the function returns `ERR_MEM`, the application should wait until some of the currently enqueued data has been successfully received by the other host and try again. This can be achieved with a callback function previously provided to `tcp_sent()`.

tcp_write()

tcp_sent()

Name

`tcp_sent()` — Set callback for successful transmission

Synopsis

```
void tcp_sent(struct tcp_pcb *pcb, err_t (*sent) (void *arg, struct tcp_pcb *tpcb,  
u16_t len));
```

Description

Specifies the callback function that should be called when data has successfully been received (i.e. acknowledged) by the remote host. The *len* argument passed to the *sent* callback function gives the number of bytes that were acknowledged by the last acknowledgment.

tcp_sent()

Receiving TCP data

TCP data reception is callback based - an application specified callback function is called when new data arrives. When the application has taken the data, it has to call the `tcp_recved()` function to indicate that TCP can advertise an increase in the receive window.

tcp_recv()

Name

`tcp_recv()` — Set callback for incoming data

Synopsis

```
void tcp_recv(struct tcp_pcb *pcb, err_t (*recv) (void *arg, struct tcp_pcb *tpcb,  
struct pbuf *p, err_t err));
```

Description

Sets the callback function that will be called when new data arrives on the connection associated with *pcb*. The callback function will be passed a `NULL` pbuf to indicate that the remote host has closed the connection.

tcp_recv()

tcp_recved()

Name

`tcp_recved()` — Indicate receipt of data

Synopsis

```
void tcp_recved(struct tcp_pcb *pcb, u16_t len);
```

Description

This function must be called when the application has received the data. *len* indicates the length of the received data.

tcp_recved()

Application polling

When a connection is idle (i.e., no data is either transmitted or received), lwIP will repeatedly poll the application by calling a specified callback function. This can be used either as a watchdog timer for killing connections that have stayed idle for too long, or as a method of waiting for memory to become available. For instance, if a call to `tcp_write()` has failed because memory wasn't available, the application may use the polling functionality to call `tcp_write()` again when the connection has been idle for a while.

tcp_poll()

Name

`tcp_poll()` — Set application poll callback

Synopsis

```
void tcp_poll(struct tcp_pcb *pcb, u8_t interval, err_t (*poll) (void *arg, struct tcp_pcb *tpcb));
```

Description

Specifies the polling interval and the callback function that should be called to poll the application. The interval is specified in number of TCP coarse grained timer shots, which typically occurs twice a second. An interval of 10 means that the application would be polled every 5 seconds.

tcp_poll()

Closing connections, aborting connections and errors

tcp_close()

Name

`tcp_close()` — Close the connection

Synopsis

```
err_t tcp_close(struct tcp_pcb *pcb);
```

Description

Closes the connection. The pcb is deallocated by the TCP code after a call to `tcp_close()`.

Return value

The function may return `ERR_MEM` if no memory was available for closing the connection. If so, the application should wait and try again either by using the acknowledgment callback or the polling functionality. If the close succeeds, the function returns `ERR_OK`.

tcp_close()

tcp_abort()

Name

`tcp_abort()` — Abort the connection

Synopsis

```
void tcp_abort(struct tcp_pcb *pcb);
```

Description

Aborts the connection by sending a RST (reset) segment to the remote host. *pcb* is deallocated. This function never fails.

If a connection is aborted because of an error, the application is alerted of this event by the callback previously registered with `tcp_err()`. Errors that might abort a connection are when there is a shortage of memory.

tcp_abort()

tcp_err()

Name

tcp_err() — Set callback for errors

Synopsis

```
void tcp_err(struct tcp_pcb *pcb, void (*err) (void *arg, err_t err));
```

Description

Set callback function to be used on connection errors. The error callback function does not get the connection's pcb passed to it as a parameter since the pcb may already have been deallocated.

tcp_err()

Lower layer TCP interface

TCP provides a simple interface to the lower layers of the system. During system initialization, the function `tcp_init()` has to be called before any other TCP function is called. When the system is running, the two timer functions `tcp_fasttmr()` and `tcp_slowtmr()` must be called at regular intervals. The `tcp_fasttmr()` should be called every `TCP_FAST_INTERVAL` milliseconds (defined in `tcp.h`, and currently 250ms) and `tcp_slowtmr()` should be called every `TCP_SLOW_INTERVAL` milliseconds, currently 500ms.

UDP interface

The UDP interface is similar to that of TCP, but due to the lower level of complexity of UDP, the interface is significantly simpler.

`udp_new()`

Name

`udp_new()` — Create a new UDP pcb

Synopsis

```
struct udp_pcb *udp_new(void);
```

Description

Creates a new connection identifier (PCB) which can be used for UDP communication. The PCB is not active until it has either been bound to a local address or connected to a remote address.

Return value

Returns the new PCB. If memory is not available for creating the new PCB, `NULL` is returned.

udp_new()

udp_remove()

Name

udp_remove() — Remove a UDP pcb

Synopsis

```
void udp_remove(struct udp_pcb *pcb);
```

Description

Removes and deallocates *pcb*.

udp_remove()

udp_bind()

Name

udp_bind() — Bind PCB to local IP address and port

Synopsis

```
err_t udp_bind(struct udp_pcb *pcb, struct ip_addr *ipaddr, u16_t port);
```

Description

Binds *pcb* to the local address indicated by *ipaddr* and port indicated by *port*. *ipaddr* can be `IP_ADDR_ANY` to indicate that it should listen to any local IP address. Port may be 0 for any port.

Return value

This function can return `ERR_USE` if all usable UDP dynamic ports are used (only relevant if *port* is 0. Otherwise `udp_bind()` will always return `ERR_OK`.

udp_bind()

udp_connect()

Name

`udp_connect()` — Set remote UDP peer

Synopsis

```
err_t udp_connect(struct udp_pcb *pcb, struct ip_addr *ipaddr, u16_t port);
```

Description

Sets the remote end of *pcb*. This function does not generate any network traffic, but only sets the remote address of the *pcb*.

Return value

This function can return `ERR_USE` if all usable UDP dynamic ports are used. Otherwise `udp_connect()` will always return `ERR_OK`.

udp_connect()

udp_disconnect()

Name

`udp_disconnect()` — Set remote UDP peer

Synopsis

```
void udp_disconnect(struct udp_pcb *pcb);
```

Description

Remove the remote end of *pcb*. This function does not generate any network traffic, but only removes the remote address of the *pcb*.

udp_disconnect()

udp_send()

Name

udp_send() — Send UDP packet

Synopsis

```
err_t udp_send(struct udp_pcb *pcb, struct pbuf *p);
```

Description

Sends the pbuf *p* to the remote host associated with *pcb*. The pbuf is not deallocated.

Return value

This function returns `ERR_OK` on success; but may return `ERR_MEM` if there is insufficient memory to prepend a UDP header, or `ERR_RTE` if no suitable outgoing network interface could be found to route the packet on.

udp_send()

udp_recv()

Name

udp_recv() — Set callback for incoming UDP data

Synopsis

```
void udp_recv(struct udp_pcb *pcb, err_t (*recv) (void *arg, struct udp_pcb *upcb,  
struct pbuf *p, struct ip_addr *addr, u16_t port), void *recv_arg);
```

Description

Registers a callback function *recv* with the PCB *pcb* so that when a UDP datagram is received, the callback is invoked. The callback argument *arg* is set as the argument *recv_arg* to *udp_recv*(). The received datagram packet buffer is held in *p*. The source address of the datagram is provided in *addr*, and the source port in *port*. The callback is expected to free the packet.

udp_recv()

System initialisation

When performing manual initialisation of lwIP for use with the raw API, a variety of lwIP functions need to be called. The exact functions will depend on the intended configuration, but the following provides an example based on a single Ethernet netif, UDP, TCP, IPv4 and DHCP.

In this example, these functions must be called in the order of appearance:

`stats_init()`

Clears the structure where runtime statistics are gathered.

`sys_init()`

Not generally used with raw API, but can be called for ease of compatibility if using sequential API in addition, initialised manually.

`mem_init()`

Initializes the dynamic memory heap defined by the CDL configuration option `CYGNUM_LWIP_MEM_SIZE`.

`memp_init()`

Initializes the memory pools defined by the CDL configuration options `CYGNUM_LWIP_MEMP_NUM_*`.

`pbuf_init()`

Initializes the pbuf memory pool defined by the CDL configuration option `CYGNUM_LWIP_PBUF_POOL_SIZE`.

`etharp_init()`

Initializes the ARP table and queue. Note: you must call `etharp_tmr` at a `ARP_TMR_INTERVAL` (by default, 5 seconds) regular interval after this initialization.

`ip_init()`

Doesn't do much at present - it should be called to handle future changes.

`udp_init()`

Clears the UDP PCB list.

`tcp_init()`

Clears the TCP PCB list and clears some internal TCP timers. Note: [as mentioned earlier](#), you must call `tcp_fasttmr()` and `tcp_slowtmr()` at the predefined regular intervals after this initialization.

```
struct netif *netif_add(struct netif *netif, struct ip_addr *ipaddr, struct ip_addr
*netmask, struct ip_addr *gw, void *state, err_t (* init)(struct netif *netif), err_t
(* input)(struct pbuf *p, struct netif *netif))
```

Adds your network interface to the `netif_list`. Allocate a struct `netif` and pass a pointer to this structure as the first argument. Give pointers to cleared struct `ip_addr` structures when using DHCP, or fill them with sane numbers otherwise. The state pointer may be NULL.

The `init` function pointer must point to a initialization function for your ethernet netif interface. The following code illustrates its use:

```
err_t netif_if_init(struct netif *netif)
{
    u8_t i;

    for(i = 0; i < 6; i++)
        netif->hwaddr[i] = some_eth_addr[i];
    init_my_eth_device();
    return ERR_OK;
}
```

The input function pointer must point to the lwIP function `ip_input()`.

```
netif_set_default(struct netif *netif)
```

Registers *netif* as the default network interface.

```
netif_set_up(struct netif *netif)
```

When *netif* is fully configured, this function must be called to allow it to be used.

```
dhcp_start(struct netif *netif)
```

Creates a new DHCP client for this interface on the first call. Note: you must call `dhcp_fine_tmr()` and `dhcp_coarse_tmr()` at the predefined regular intervals after starting the client.

You can peek in the `netif->dhcp` struct for the actual DHCP status.

VI. Object Loader

Object Loader

Name

CYGPKG_OBJLOADER — eCos Support for Dynamic Module Loading

Synopsis

```
#include <cyg/objloader/objload.h>

void *cyg_ldr_open(cyg_ldr_open_stream *open_stream, CYG_ADDRWORD data);
void cyg_ldr_close(void *handle);
char *cyg_ldr_error(void);
void *cyg_ldr_find_symbol(void *handle, char *symbol);
```

Description

Note: The Object Loader package does not support all processor architectures at present.

The Object Loader package provides support for dynamically loading executable modules into an eCos system. Modules may be loaded into memory from a variety of sources, linked in to the running system and entry points invoked to execute the code of the module. When the module is no longer required, it may be unloaded and the memory reused for other purposes or other modules.

This system is modelled most closely on the Linux kernel module mechanism, rather than Windows DLLs or Unix shared objects. As a result, it has a number of restrictions:

- Only modules written in C are supported. The Object Loader does not currently provide support for invoking static constructors and destructors, C++ exceptions, RTTI and other parts of the C++ runtime system.
- Automatic symbol resolution only works for references from a module into the main executable. References between modules are not supported, and resolution of unresolved symbols in the main executable to module symbols is not supported.
- Loaded modules need to be built using the same, or similar, configuration to the main system.
- Loaded modules should be built with the same or compatible compiler flags as the main system. There is one important exception. Some architectures including MIPS and Nios II implement a global pointer register. Small global variables are placed in an area of memory up to 64K. The gp register points at this area of memory, allowing the variables to be accessed directly using a single instruction instead of the two or more instructions that would otherwise be required. This technique cannot be used for a dynamically loaded module. Hence the use of gp-relative addressing must be suppressed with a compiler flag, typically `-G0`.

Creating Loadable Modules

Modules can be just object files as generated by the compiler. In a Makefile including the `$(INSTALL_DIR)/include/pkgconf/ecos.mak` definitions file, the entry to build `module.o` might be:

```
module.o: module.c
    $(ECOS_COMMAND_PREFIX)gcc -c -I$(INSTALL_DIR)/include $(ECOS_GLOBAL_CFLAGS) -o $@ $<
    $(ECOS_COMMAND_PREFIX)strip -g $@
```

The compile line generates a `.o` file. The `-I` option allows includes to be fetched from the eCos installation. The command prefix and global flags are stored in the `ecos.mak` file by the eCos build process. If the compile flags include `-g` or some other debug option then to save memory and maybe load time it is useful to pass the finished module through **strip** to limit the file contents to just the loadable ELF sections.

It is possible to create a module out of several object files by using the linker's ability to perform a partial link:

```
module.o : file1.o file2.o file3.o
    $(ECOS_COMMAND_PREFIX)gcc $(subst --gc-sections,-r,$(ECOS_GLOBAL_LDFLAGS)) -L$(PREFIX)
    $(ECOS_COMMAND_PREFIX)strip -g $@
```

The `module.ld` linker script is defined by the Object Loader package and is copied out to the install lib directory. It should be used when combining multiple files, or when advanced features such as HAL tables are used in a single object file.

If the module makes use of float, double, long long and some long arithmetic operations, then it should be partially linked against `libgcc` before loading. This can be done with the following makefile fragments:

```
# Single source file module...
module.o: module.c
    $(ECOS_COMMAND_PREFIX)gcc -c -I$(INSTALL_DIR)/include $(ECOS_GLOBAL_CFLAGS) -o $@.tmp
    $(ECOS_COMMAND_PREFIX)gcc $(subst --gc-sections,-r,$(ECOS_GLOBAL_LDFLAGS)) -L'dirname
    $(ECOS_COMMAND_PREFIX)strip -g $@

# Combine multiple object files...
module.o : file1.o file2.o file3.o
    $(ECOS_COMMAND_PREFIX)gcc $(subst --gc-sections,-r,$(ECOS_GLOBAL_LDFLAGS)) -L'dirname
    $(ECOS_COMMAND_PREFIX)strip -g $@
```

Target Specific Considerations

There are a number of special considerations for particular target architectures:

- Modules compiled for Thumb may be loaded into targets compiled for either ARM32 or Thumb. Thumb builds of eCos that use the object loader should have the `"-mlong-calls"` compiler option set. ARM32 builds should have thumb interworking enabled if thumb modules are to be loaded (the object loader module does this automatically). Thumb modules should be compiled with `"-mthumb -mthumb-interwork -mlong-calls"` compiler options.
- Modules compiled for the MIPS16 instruction set may be loaded into a MIPS target, so long as the processor supports the instruction set. To compile and link such a module, the `"-mips16"` compiler option must be substituted for `"-mips32"`, along with `"-fwritable-strings"`.

Loading Modules

The function `cyg_ldr_open()` is used to load a module into memory. It takes two arguments. The first argument defines a module loader, while the second argument is a generic data item whose value depends on the loader. If the load is successful, then a non-NULL handle will be returned. A NULL pointer will be returned on failure.

If there is an error in the loading process, then the function `cyg_ldr_error()` will return a string describing the last error that occurred. Note that this is not thread-safe since there is only a single last error recorded for all load operations.

At present the following loaders are implemented:

CYG_LDR_FILESYSTEM

This loader uses FILEIO operations to read an ELF file from a named file in a filesystem. For example, to read a module from the file `"/lib/modules/module.o"`:

```
mod_handle = cyg_ldr_open( CYG_LDR_FILESYSTEM, (CYG_ADDRWORD)"/lib/modules/module.o");
```

This loader is included by default if the `CYGPKG_IO_FILEIO` package is included, although it can be omitted by disabling `CYGPKG_OBJLOADER_LOADER_FS`.

CYG_LDR_MEMORY

This loader uses memory access primitives to read an ELF file from any addressable memory such as ROM, FLASH or RAM. For example to read a module from the location `module_base`:

```
mod_handle = cyg_ldr_open( CYG_LDR_MEMORY, (CYG_ADDRWORD)&module_base);
```

This loader is included by default, although it can be omitted by disabling `CYGPKG_OBJLOADER_LOADER_MEM`.

Loaders `CYG_LDR_FTP`, `CYG_LDR_TFTP`, `CYG_LDR_HTTP` and `CYG_LDR_FLASH` are defined, but not currently implemented.

Unloading Modules

A module may be unloaded by calling `cyg_ldr_close()`, passing it the handle returned from `cyg_ldr_open()`. This will cause the memory occupied by the loader to be released. Any pointers into the code or data of the module will be rendered invalid and should not be used.

Referencing Module Symbols

When a module is loaded, a symbol table listing all the external symbols that it defines is loaded with it. The function `cyg_ldr_find_symbol()` searches this table and returns a pointer to the location defined by a symbol. For example, to create a thread running from a function in a module:

```
cyg_thread_entry_t *thread_entry;

thread_entry = cyg_ldr_find_symbol( handle, "thread_entry");

cyg_thread_create(THREAD_PRIORITY,
```

```

        thread_entry,
        0,
        "Module Thread",
        (void *)thread_stack,
        THREAD_STACK_SIZE,
        &thread_handle,
        &thread_object);

```

Both functions and variables may be accessed in this way.

There is no mechanism for resolving dangling references in the main eCos application, or other modules, to symbols in a newly loaded module. The main eCos application must have all references resolved at link time. However, it is possible to simulate the effect of dynamic resolution by using function pointers. For example define a global function pointer to an initial dummy function:

```

typedef int module_fn_t(int a, int b);

module_fn_t dummy_fn;

module_fn_t *module_fn = dummy_fn;

int dummy_fn( int a, int b )
{
    return -1;
}

```

When the module is loaded the function pointer can be pointed at the function within the module, and pointed back to the dummy function when it is unloaded:

```

void *mod_handle;

void load_module(void)
{
    mod_handle = cyg_ldr_open( CYG_LDR_FILESYSTEM, (CYG_ADDRWORD)"/lib/modules/module.o");

    module_fn = cyg_ldr_find_symbol( mod_handle, "module_fn");
}

void unload_module(void)
{
    cyg_ldr_close( mod_handle );

    module_fn = dummy_fn;
}

```

One could even implement a form of demand loading by combining `dummy_fn` and `load_module`:

```

int dummy_fn( int a, int b )
{
    mod_handle = cyg_ldr_open( CYG_LDR_FILESYSTEM, (CYG_ADDRWORD)"/lib/modules/module.o");

    module_fn = cyg_ldr_find_symbol( mod_handle, "module_fn");
}

```



```

    return module_fn( a, b );
}

```

Module Open and Close Functions

When a module is loaded the Object Loader will look for a symbol with the name "module_open" and if found will call it with the following prototype:

```
void module_open( void );
```

Similarly, when `cyg_ldr_close()` is called, the Object Loader will look for a symbol named "module_close" and call it, with the same prototype.

External References

When a module is loaded, the Object Loader package performs any relocations it requires and resolves any unresolved symbol references it contains. The load only succeeds if all of these can be completed. For these symbols to be resolved it is necessary for the main eCos executable to contain a symbol table defining the symbols to be resolved. Normal eCos executables do not contain such a symbol table since it would occupy an unreasonably large amount of memory. There is also no mechanism to persuade the linker to include a loadable symbol table into the executable. Hence it is necessary for the application to explicitly define a symbol table that maps symbol names to addresses.

The loader provides an empty table; the user can then define additional entries required by any loadable modules. In order to keep the size of the table to a minimum, the user can selectively include only those functions that are expected to be used by the loader to resolve all references. There are several macros defined in `objload.h` for defining table entries:

```
CYG_LDR_TABLE_FUN( name )
```

This macro defines a table entry for a function with the given name.

```
CYG_LDR_TABLE_VAR( name )
```

This macro defines a table entry for a data variable with the given name.

```
CYG_LDR_TABLE_ENTRY( entry_name, symbol_name, address )
```

This is a low level macro that allow all aspects of a symbol table entry to be controlled. The *entry_name* argument defines the table entry object name (a C language requirement since anonymous objects are not permitted). The *symbol_name* argument is a string giving the symbol that will be matched by the loader. The *address* argument gives the memory location to which this symbol will resolve.

The `objload.h` file contains a number of macros that collect together groups of functions as a convenient way to include blocks of Kernel, C Library and FILEIO functionality. These include the following:

```

CYG_LDR_TABLE_KAPI_ALARM( )
CYG_LDR_TABLE_KAPI_CLOCK( )
CYG_LDR_TABLE_KAPI_COND( )
CYG_LDR_TABLE_KAPI_COUNTER( )
CYG_LDR_TABLE_KAPI_EXCEPTIONS( )

```

Object Loader

```
CYG_LDR_TABLE_KAPI_FLAG( )  
CYG_LDR_TABLE_KAPI_INTERRUPTS( )  
CYG_LDR_TABLE_KAPI_MBOX( )  
CYG_LDR_TABLE_KAPI_MEMPOOL_FIX( )  
CYG_LDR_TABLE_KAPI_MEMPOOL_VAR( )  
CYG_LDR_TABLE_KAPI_MUTEX( )  
CYG_LDR_TABLE_KAPI_SCHEDULER( )  
CYG_LDR_TABLE_KAPI_SEMAPHORE( )  
CYG_LDR_TABLE_KAPI_THREAD( )  
CYG_LDR_TABLE_STRING( )  
CYG_LDR_TABLE_STDIO( )  
CYG_LDR_TABLE_INFRA_DIAG( )  
CYG_LDR_TABLE_FILEIO( )  
CYG_LDR_TABLE_NET( )
```

Extending the Object Loader

Name

CYGPKG_OBJLOADER — Extending the Object Loader

Description

The Object Loader package has a number of features that allow it to be extended. To support a new CPU architecture a new relocater needs to be written. If ELF files are to be read from a source that differs from those currently supported, then a new loader needs to be written. Finally, the mechanism by which the loader allocates the memory used to store loaded sections can be redirected by the application.

Adding New Relocators

When the loader loads a new module some locations in it must be adjusted to account for the address at which is it loaded. References to external symbols must also be installed. The location and nature of these modifications are described by one or more sections in the ELF file which contain a sequence of relocation records. The exact meaning of the relocations that these records define is architecture specific and is usually described as part of the ABI for that CPU type.

To define a new relocater for a CPU, it is necessary to add an extra definition to the `objloader.cdl` file, and add a header and source file to the package. To support the XYZ CPU the following must be added to the `CYGPKG_OBJLOADER_ARCHITECTURE` component:

```
cdl_option CYGBLD_OBJLOADER_ARCHITECTURE_XYZ {
    display      "Support loading on XYZ processors"
    calculated    CYGPKG_HAL_XYZ
    implements    CYGINT_OBJLOADER_RELOCATOR
    define_proc {
        puts $::cdl_header "#include <cyg/objloader/relocate_xyz.h>"
    }
    compile relocate_xyz.c
}
```

The `relocate_xyz.h` file needs to define some macros to customize the loader:

`ELF_ARCH_MACHINE_TYPE`

This defines the value that the `e_machine` machine type field of the ELF header. If this does not match, the module load will fail.

`ELF_ARCH_ENDIANNES`

This defines the value that the `EI_DATA` byte of the `e_ident` field of the ELF header. It should be either `ELFDATA2LSB` or `ELFDATA2MSB`. If this does not match, the module load will fail. Architectures that are bi-endian need to test either a compiler or a HAL definition to select the correct endianness for the current build.

```
CYG_LDR_MAKE_LOCAL_ADDRESS( addr, sym )
```

This macro is used to combine the section address of a symbol with information from its symbol table entry. The *addr* argument is the base address of the section in which the symbol is defined. The *sym* is the symbol table entry; this is not a pointer, so fields should be accessed using the "." operator, not the "->" operator. The return value should be of type void *. This is an optional macro, if it is not defined here then a default definition will be used which simply adds the *st_value* field of the symbol table entry to the address.

In addition to these, it may also contain definitions that are useful to the relocater. Typically the relocation record types and any support macros may be defined here.

The *relocate_xyz.c* file contains two functions:

```
void cyg_ldr_flush_cache(void)
```

This function is called to perform any cache flushing needed. The loader modifies code in memory that will subsequently be executed. It does this using data accesses, so it is essential that these updates are flushed from the data cache, and that stale entries are flushed from the instruction cache. This function must call appropriate HAL cache operations to ensure that this is done.

```
cyg_int32 cyg_ldr_relocate(cyg_int32 rel_type, cyg_uint32 flags, cyg_uint32 mem,  
cyg_int32 sym_value)
```

The loader will call this function for each relocation record in each relocation section found in the module.

The *rel_type* argument defines the relocation record type and will be one of the relocation types defined for the architecture. Most architecture ABIs define a large number of relocations, not all of which will be relevant to the use that eCos makes of the object file format. In general only a small subset of relocation types need to be handled which can usually be determined by inspecting the object files generated by compiling eCos.

The *flags* argument contains flags that provide additional information about the relocation record. At present only one flag is defined: *CYG_LDR_FLAG_RELA* which is set when the relocation is from a RELA record, otherwise it comes from a REL record.

The *mem* argument contains the address of the location in memory to be relocated. It is constructed from the base address of the segment targeted by the relocation section, plus the *r_offset* field from the relocation record.

The *sym_value* argument contains the address of any symbol associated with the relocation record. If it was a RELA record, then the contents of the *r_addend* field will have been added.

Adding new Loaders

The Object Loader package needs to fetch a module from some source to load it into memory. This is the job of a loader. A loader consists of an open function plus read, seek and close functions.

The loader open function is supplied as the first parameter to *cyg_ldr_open()*. It is called with the second argument as a parameter. On success it returns a pointer to a *CYG_LDR_ELF_OBJECT* object. On failure it returns NULL.

The open function has a number of duties, best described by an annotated example for the ABC loader:

```

__externC CYG_LDR_ELF_OBJECT *cyg_ldr_open_abc(CYG_ADDRWORD arg)
{
    // Allocate a CYG_LDR_ELF_OBJECT
    CYG_LDR_ELF_OBJECT * obj = (CYG_LDR_ELF_OBJECT *)cyg_ldr_malloc(sizeof(CYG_LDR_ELF_OBJECT))

    // Allocate a private data descriptor. Depending on the nature of
    // the loader this may not be necessary.
    struct abc_desc *desc = cyg_ldr_malloc(sizeof(struct abc_desc));

    // Check that the memory allocations worked
    if( obj == NULL || desc == NULL )
    {
        if( obj != NULL ) free(obj);
        if( desc != NULL ) free(desc);
        cyg_ldr_last_error = "ERROR IN MALLOC";
        return NULL;
    }

    // Perform any operations to enable access to the ELF file and
    // fill in the descriptor. If this fails then free both descriptor
    // and ELF object, set cyg_ldr_last_error and return NULL.

    // Clear the CYG_LDR_ELF_OBJECT
    memset( obj, 0, sizeof(CYG_LDR_ELF_OBJECT));

    // Install private data pointer
    obj->ptr = (CYG_ADDRWORD)desc;

    // Install pointers to read, seek and close functions
    obj->read = cyg_ldr_abc_read;
    obj->seek = cyg_ldr_abc_seek;
    obj->close = cyg_ldr_abc_close;

    // Return completed object
    return obj;
}

```

The read function will be called via the pointer in the ELF object whenever the Object Loader needs to read data from the file. It has the following definition:

```
static size_t cyg_ldr_abc_read(struct CYG_LDR_ELF_OBJECT* obj, size_t size, void* buf)
```

The *obj* argument is the ELF object returned from the open function. The *size* argument gives the number of bytes to be read and *buf* points to a location to store them. The function returns the number of bytes read.

The seek function will be called via the pointer in the ELF object whenever the Object Loader needs to reposition the point in the file at which the next read will occur. It has the following definition:

```
static cyg_int32 cyg_ldr_abc_seek(struct CYG_LDR_ELF_OBJECT* obj, cyg_uint32 offset)
```

The *obj* argument is the ELF object returned from the open function. The *offset* argument gives the number of bytes from the start of the file to which the read point should be moved. The function returns the new read offset.

Some sources may not be able to reposition the read pointer backwards, and may only be capable of advancing it. If the reposition fails then this function should return -1.

The close function will be called via the pointer in the ELF object when the Object Loader has finished with the file. It has the following definition:

```
static cyg_int32 cyg_ldr_abc_close(struct CYG_LDR_ELF_OBJECT* obj)
```

The *obj* argument is the ELF object returned from the open function. This function should close down access to the file, free the private data descriptor if necessary and set the *obj->ptr* field to zero. It should not free the ELF object itself, the Object Loader will do this itself later. If the close succeeds then this function should return zero, and -1 if it fails.

Redirecting Memory Allocation

All memory allocation in the Object Loader is made via the `cyg_ldr_malloc()` function and it is freed via the `cyg_ldr_free()` function. These have the following prototypes:

```
__externC void *cyg_ldr_malloc(size_t) CYGBLD_ATTRIB_WEAK;
```

```
__externC void cyg_ldr_free(void *) CYGBLD_ATTRIB_WEAK;
```

These functions by default simply call the standard `malloc()` and `free()` heap functions. However, they are defined with the *weak* linker attribute. This means that the application can redefine these functions to provide an alternative allocation and free mechanism if, for example, the standard heap support has been omitted.

VII. The eCos NAND Flash Library

Chapter 6. NAND Library Overview

Description

This is a library which allows NAND flash devices to be accessed by the eCos kernel and applications. It is analogous to the eCos FLASH library, but for NAND devices. It exists as a separate library because of the fundamental differences between the two types of flash memory.

This library provides the following functionality:

- Interrogation to confirm that the expected device is present
- Reading from and writing to flash pages
- Erasing flash blocks
- The ability to divide a single device into multiple partitions, like those of a hard drive
- Creation and maintenance of a Bad Block Table
- Use of an Error Correcting Code to detect and correct single-bit errors, and to detect multiple-bit errors
- Packing of the ECC and application out-of-band data into the spare area on the device

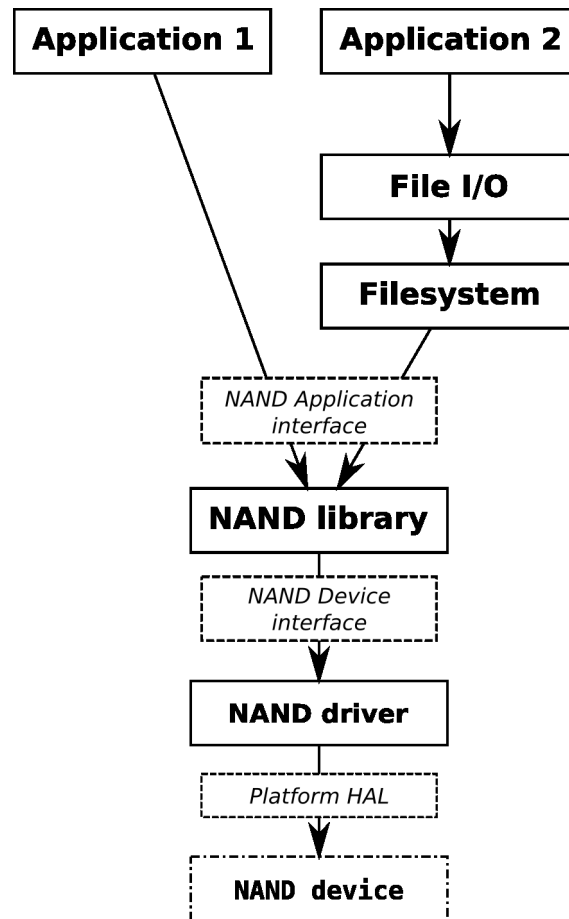
Note: The spare area, ECC and bad block table have been deliberately created with the intention of compatibility with current versions of the Linux MTD layer. For example, this would allow a single NAND device to be accessed by RedBoot to load a Linux kernel, which could then go on to use another partition as its root filesystem.

Tip: This library is also used as glue to allow appropriate filesystems to use NAND devices. This allows more useful higher-level access by applications and RedBoot via the File I/O and POSIX interfaces. In other words, your application may not need to invoke this library directly, though of course you may still have to write a driver for your chip and/or board.

Structure of the library

This library has two principal interfaces: one for *applications* to call into it, and another to call out to the chip-specific *drivers*. (The chip drivers themselves then require support from the relevant platform HAL to allow them to access the physical chip in an appropriate manner for the board - such as the memory-mapped I/O range to use.)

The following diagram illustrates the calls from two applications all the way to an underlying NAND device. Application 1 uses the NAND library directly, whereas application 2 is using a filesystem and the eCos File I/O layer.



Device support

Before this library can be used on a given board, an appropriate device driver must be created. Each driver is for a particular NAND part or family of parts; the HAL for each board then instantiates the relevant driver(s) appropriately with board-specific glue such as the memory-mapped I/O range to use. Full details on creating a driver are presented in [Chapter 8](#).

There is also a [Synthetic Target NAND Flash Device](#) for testing purposes, which is present on the *synth* target.

Danger, Will Robinson! Danger!

Unlike nearly every other aspect of embedded system programming, getting it wrong with FLASH devices can render your target system useless. Most targets have a boot loader in the FLASH. Without this boot loader the target will obviously not boot. So before starting to play with this library its worth investigating a few things. How do you recover your target if you delete the boot loader? Do you have the necessary JTAG cable? Or is specialist hardware needed? Is it even possible to recover the target boards or must it be thrown into the rubbish bin? How does killing the board affect your project schedule?

Differences between NAND and NOR flash

Most flash devices supported by the eCos Flash library are categorised as NOR flash. These are fundamentally different from NAND flash devices, both in terms of the storage cells deep within the chip, and how they are addressed and used by applications.

Attribute	NOR	NAND
Addressing of data	By byte address within the device. Usually expressed as memory-mapped addresses.	By row (page) number. Pages are a power of two; commonly 512 or 2048 bytes. Optimised for reading and writing a page at a time. Sometimes supports column (byte) addressing, but this library does not expose such functionality.
Are direct reads and writes possible? ^a	Usually	Not in general, though a few special-case exceptions exist such as OneNAND devices.
Erase block size	May vary across the chip	A fixed number of pages, typically 64
Out-of-band data	Not supported	A small number of bytes per page - typically 16 "spare" per 512 "data" bytes - are usable by the application. They are read, written and erased at the same time as the "real" page data.
May factory-bad regions ^b exist on the chip?	No	Typically up to 20 eraseblocks are marked as factory-bad in their OOB area. The OS is expected to scan these to create a Bad Block Table. ^c
May data be rewritten without being erased first?	Usually (but only by resetting 1-bits to 0)	Usually, on SLC NAND chips; not on MLC chips.
Error detection and correction	Not present	Usually automatic. Typically this involves an Error Correcting Code, automatically calculated and stored in the OOB area, then checked on read.

Notes:

- a. In other words, can the application read flash directly as if it was RAM, or does it have to invoke the driver to copy data in and out?
- b. Regions which were found during manufacture to be bad and marked in some way - usually by placing a special code in the Out Of Band area.
- c. Once a BBT exists it can then be used to keep track of any blocks which fail through wear during the lifetime of the device.

Since a NAND chip can in general only be read indirectly, its contents must be copied to RAM before they can be executed. This means that the caveats in the eCos FLASH library about disabling interrupts whilst programming do not apply here, except in special cases such as OneNAND devices.

Preparing for deployment

It is generally not recommended to hard-code physical on-NAND locations in case of factory bad blocks or block failures in the field. ¹ Instead it is preferable to set up *partitions* on the chip with a generous safety margin and to store data in a location-independent way. This is commonly achieved by placing logical tags in the spare area of each page, or using a log-structured filesystem such as YAFFS. Such strategies remove the dependence on physical addressing, at the cost of increased complexity.

The upshot of this is that you cannot reliably create a simple binary image to bulk-program in the factory. A more complicated programming operation is required to take account of your chip partitions, logical addressing strategy and any bad blocks which may be encountered during write.

Notes

1. Usually the first block is guaranteed to be defect free for a certain number of erase cycles. This tends to be necessary if bootstrapping the CPU off NAND, and is an obvious exception to this rule.

Chapter 7. Using the NAND library

The eCos NAND library exposes two principal APIs: one for applications to use and the other to communicate with device drivers.

Configuring the NAND library

The following configuration options are provided. They affect the library globally, i.e. across all drivers.

`CYGPKG_IO_NAND_CFLAGS_ADD`

`CYGPKG_IO_NAND_CFLAGS_REMOVE`

Allows specific build options to be added to or removed from the CFLAGS list when building this library.

`CYGSEM_IO_NAND_DEBUG`

This is the master switch for all debug reporting from the library.

`CYGSEM_IO_NAND_DEBUG_FN_DEFAULT`

This is the default function that the library will use when sending debugging output. It must behave like `printf`. The default - `cyg_nand_defaultprintf` - is a wrapper to `diag_printf`.

Note: Individual drivers may override this setting in their `devinit` routines by overwriting the pointer in the device struct.

`CYGSEM_IO_NAND_DEBUG_LEVEL`

Specifies the verbosity of the NAND library and device drivers. Ranges from 0 (off) to 9 (incredibly verbose); the default setting is 1. (Higher values are only likely to be of use during driver development, if ever.) When enabled, messages are printed using the per-device `printf`-like function (see above).

Note: Should a serious problem be encountered it will always be reported the `printf`-like function, regardless of this setting. Such messages may be suppressed altogether by turning off `CYGSEM_IO_NAND_DEBUG`.

`CYGSEM_IO_NAND_READONLY`

Globally disables all code which writes to NAND devices. This may be useful during driver development.

`CYGNUM_NAND_MAX_PARTITIONS`

Sets a compile-time limit on the number of partitions any NAND device may have. The default is 4, which should be enough for most purposes; unnecessarily setting this higher wastes RAM.

CYGSEM_IO_NAND_USE_BBT

Globally enables and disables the use of Bad Block Table.

Warning

This setting should not be disabled lightly! It is strongly recommended that you leave this setting enabled unless you have a very good reason to not use it. It is provided really as a convenience for allowing developers to recover their NAND from a confused state.

The NAND Application API

All of the functions described here are declared in the header file `<cyg/nand/nand.h>`, which should be included by all users of the NAND library.

Note: Most of the functions in the library are declared as returning `int`. *Unless otherwise stated, all functions return 0 for success, or a negative eCos error code if something went wrong.*

Device initialisation and lookup

NAND devices are identified to the library by name. In many cases there will be only one, commonly named *onboard*, but this flexibility allows for easy expansion later without cross-device confusion.

Note: The naming of NAND devices is set up by the code that instantiates their drivers. Normally this is done by the platform HAL port.

```
__externC int cyg_nand_lookup(const char *devname, cyg_nand_device **dev_o);
```

On success, `*dev_o` will be set up to point to a `cyg_nand_device` struct. On failure, it will not; a return code of `-ENOENT` signifies that the requested device name was not found.

Applications will hardly, if ever, need to access the `cyg_nand_device` structs directly. The following members and convenience macros are most likely to be of relevance:

```
struct _cyg_nand_device_t {
    ...
    cyg_nand_printf pf; // Diagnostic printf-like function for this device to use. May be changed at r
    ...
    size_t page_bits; // log2 of no of regular bytes per page
    size_t spare_per_page; // OOB area size in bytes
    size_t block_page_bits; // log2 of no of pages per eraseblock
    size_t blockcount_bits; // log2 of number of blocks
    size_t chipsize_log; // log2 of total chip size in BYTES.
    ...
};
```

```
#define CYG_NAND_BYTES_PER_PAGE(dev) (1<<(dev)->page_bits)
#define CYG_NAND_SPARE_PER_PAGE(dev) ((dev)->spare_per_page)
#define CYG_NAND_PAGES_PER_BLOCK(dev) (1<<(dev)->block_page_bits)
#define CYG_NAND_BLOCKCOUNT(dev) (1<<(dev)->blockcount_bits)
#define CYG_NAND_PAGECOUNT(dev) (NAND_BLOCKCOUNT(dev) * NAND_PAGES_PER_BLOCK(dev))
#define CYG_NAND_CHIPSIZE(dev) (1<<(dev)->chipsize_log)
#define CYG_NAND_APPSPARE_PER_PAGE(dev) ((dev)->oob->app_size)
```

NAND device addressing

NAND devices are arranged as a series of *pages* and *eraseblocks*. The eCos NAND library numbers pages and eraseblocks sequentially, both starting at 0 and continuing until the end of the chip. For example, eraseblock 0 might contain pages 0 through 63; eraseblock 1, pages 64 through 127; and so on.

Caution

This numbering scheme is independent of the device's addressing scheme. Take care, particularly when erasing blocks; some devices and some applications effectively express the location to erase as a page number (or, in NAND-speak, as the *row address* to erase from).

Warning

Most NAND chip manufacturers document restrictions on the order in which pages may be written to their device. Typically, individual pages within an eraseblock must be written in sequential order starting from the first, and random-order writes are prohibited or unspecified. The eCos NAND library does not attempt to police such restrictions; if at all unsure, check the spec sheet for the part. You have been warned!

NAND devices are widely considered to be arranged as one or more *partitions*, and the eCos NAND library supports this. However, there is no universal scheme for partition sizes to be supplied to the driver, unlike hard drives which encode a partition table into their first sector. Partition arrangements are often implicitly hardcoded, such as by byte address within the device, though they could be encoded in a "partition table", user-set, or even variable under software control by some esoteric rules. Therefore, every device driver is responsible for configuring its partition information as appropriate for the device, and this might for example appear as CDL options.

Tip: Be sure to read the notes associated with the device driver to understand how partitions are set up; if no notes are provided, look in its `devinit` code.

NAND device partitions

After a NAND device has been initialised, its device struct contains a list of partitions. These are numbered from 0 and may go up to `CYGNUM_NAND_MAX_PARTITIONS-1`. Before an application can use the NAND device, it must obtain a partition context (pointer) with the following call:

```
__externC cyg_nand_partition* cyg_nand_get_partition(cyg_nand_device *dev, unsigned partno);
```

Note: This call returns a pointer to the partition struct, not an error code. If the given partition number is inactive or invalid, it returns NULL.

About the spare area

Every page on the NAND array has a small number of "spare" bytes associated with it. These are used by the NAND library to store the page's ECC; whatever is left over may be used by the application for whatever purposes may suit it.

Every page has `CYG_NAND_APPSPARE_PER_PAGE(dev)` bytes of spare area available to the application. (This amount is implicit from the driver configuration and cannot change during the lifetime of a device.)

Note: Application spare bytes are not subject to the ECC. When reading the spare area data, you must be prepared to cope with the consequences of the (unlikely) event of a bit drop-out or other failure.

Manipulating the NAND array

Now, finally, given a `cyg_nand_partition*`, your application can make use of the NAND array with the following functions:

Reading data

```
__externC int cyg_nand_read_page(cyg_nand_partition *ctx, cyg_nand_page_addr page, void * dest, si
```

Reads a single page and its spare area. The data read from the chip will be automatically ECC-checked and repaired if necessary. Parameters are as follows:

ctx

The partition that data is to be read from.

page

The page to be read, *numbered from the start of the partition*.. As a double-check, the library will refuse the operation with `-ENOENT` if this address is not within partition *ctx*.

Note: This was changed in application interface v2; earlier page and block addresses were device-relative.

dest

Where to put the data. May be `NULL`, in which case the page data is not read.

size

The maximum amount of data to read. (In any event, no more than a single page will be read, but if your application knows it doesn't need the whole page, you can place a cap here.)

spare

Where to store the application data read from the spare area. This may be `NULL` if spare data is not required.

spare_size

The maximum number of bytes to read from the spare area. This will not be more than `CYG_NAND_APPSPARE_PER_PAGE(dev)` bytes.

An error response of `-EIO` means that a multiple-bit I/O error has occurred in the page data, which the ECC could not repair. The library stores the data read from the device in **dest* and **spare* on a best-effort basis; it should not be relied upon. The application should take steps to salvage what it can and erase the block as soon as possible.

Writing data

```
__externC int cyg_nand_write_page(cyg_nand_partition *ctx, cyg_nand_page_addr page, const void *s
```

Writes a single page and its spare area. The ECC will be computed and stored automatically. Parameters are as follows:

ctx

The partition that data is to be written to.

page

The page to be written, *numbered from the start of the partition*. As a double-check, the library will refuse the operation with `-ENOENT` if this address is not within partition *ctx*.

src

Where to read the data from. May be `NULL`, in which case the page data is not written.

size

The amount of data to write. (In any event, no more than a single page will be written.)

spare

Where to read the data to go into the spare area; it will automatically be packed around the ECC as necessary. Again, this may be `NULL` if spare data is not required.

spare_size

The number of bytes to write to the spare area. This should not be larger than `CYG_NAND_APPSPARE_PER_PAGE(dev)`; if it is, only that many bytes will be stored.

An error response of `-EIO` means that the page write failed. The application should copy out any data it wishes to keep from the rest of the eraseblock, then call `cyg_nand_bbt_markbad()` to put the block beyond use.

Erasing blocks

```
__externC int cyg_nand_erase_block(cyg_nand_partition *ctx, cyg_nand_block_addr blk);
```

ctx

The partition that data is to be erased from.

blk

The block to be erased, *numbered from the start of the partition*. As a double-check, the library will refuse the operation with `-ENOENT` if this address is not within partition *ctx*.

An error response of `-EIO` means that the block erase failed. In this case, the library automatically marks the block as bad, and the application need take no further action.

Common error returns

The following common error returns may be encountered when manipulating the NAND array using the above functions:

-EIO

The operation could not be completed due to an I/O error. This may require the application to take further action; check the details provided above for the call you have just made.

-ENOENT

The page or block address was not valid for the given partition.

-EINVAL

The page (block) address was (within) a block that is marked bad.

Ancillary NAND functions

The following functions are provided to allow applications to interact with the Bad Block Table:

```
typedef enum {
    CYG_NAND_BBT_OK=0,
    CYG_NAND_BBT_WORNBAD=1,
    CYG_NAND_BBT_RESERVED=2,
    CYG_NAND_BBT_FACTORY_BAD=3
} cyg_nand_bbt_status_t;

__externC int cyg_nand_bbt_query(cyg_nand_partition *ctx, cyg_nand_block_addr blk);

__externC int cyg_nand_bbt_markbad(cyg_nand_partition *ctx, cyg_nand_block_addr blk);
```

To determine the status of an eraseblock, use `cyg_nand_bbt_query`; this returns an enum from `cyg_nand_bbt_status_t` or a negative eCos error code. All blocks which return a non-0 enum value are considered inaccessible by applications.

Occasionally, it is necessary for applications to mark a block as bad. This most commonly happens when a write operation fails (see [the Section called *Writing data*](#) above). To do this, call `cyg_nand_bbt_markbad`; the return is 0 for success, or a negative eCos error code. *As with other calls, blocks are numbered from 0 at the start of the partition, and internally translated for the device as appropriate.*

Both of these calls may foreseeably return `-ENOENT` if the given block address was not valid, or `-EIO` if something awful happened with the on-chip bad block table.

Chapter 8. Writing NAND device drivers

Planning a port

Before you start, you will need to have sight of appropriate spec sheets for both the NAND chip and the board into which it is connected, and you need to know how the chip is to be partitioned.

Driver structure and layout

A typical NAND device driver falls into two parts:

- high-level operations specific to the NAND chip (page reads and writes); and
- board-specific plumbing (sending commands and data to the chip; reading data back from the chip).

This distinction is important in the interests of code reuse; the same part may appear on different boards, or indeed multiple times, but connected differently. It need not be maintained if there are good reasons not to.

The *NAND library device interface* consists of a C struct, `cyg_nand_device`, comprising a number of data fields and function pointers. Each NAND chip to be made available to the library requires exactly one instance of this struct.

Tip: The `cyg_nand_device` structure includes a `void* priv` member which is treated as opaque. The driver may use this member as it sees fit; it is intended to provide an easy means to identify the NAND array, MMIO addresses or function pointers to use and so on. Typically this is used by the chip driver for its own purposes, and includes a further opaque member for the use of the HAL port.

The function pointers in the struct form the driver's high-level functions; they make use of the low-level functions to talk to the chip. We present the high-level functions first, although there is no intrinsic reason to prefer either ordering during driver development.

The high-level chip-specific functions are traditionally laid out as an *inline file* in an appropriate package in `devs/nand/CHIP`. The board-specific functions should normally appear in the platform HAL and `#include` the inline.

Chip partitions

Before embarking on the port, you should determine how the NAND array will be partitioned. This is necessarily a board-specific question, and your layout must accommodate any other software users of the array. You will need to know either the fixed layout - converted to eraseblock addresses - or how to determine the layout at initialisation time.

Tip: It may be worthwhile to set up partitioning by way of some parameters in your platform's CDL, with sensible defaults, instead of outright hard-coding the partition layout.

Locking against concurrent access

The eCos NAND library provides per-device locking, to guard against concurrent access during high-level operations. This support is fully automatic; drivers need take no action to make use of it.

This strategy may not be sufficient on all target boards: sometimes, accessing a NAND chip requires mediation by CPLD or other device, which must be shared with other NAND chips or even other peripherals. *If this applies, it is the responsibility of the driver and platform port to provide further locking as appropriate!*

Tip: When using mutexes in a driver, one should use the *driver API* as defined in `<cyg/hal/drv_api.h>` instead of the full kernel API. This has the useful property that mutex operations are very cheaply implemented when the eCos kernel is not present, such as when operating in RedBoot.

Required CDL declarations

An individual NAND chip driver must declare the largest page size it supports by means of CDL. This is done with a statement like the following in its `cdl_package` stanza:

```
requires          ( CYGNUM_NAND_PAGEBUFFER >= 2048 )
```

Note: This requirement is due to the internal workings of the eCos NAND library: a buffer is required for certain operations which manipulate up to a NAND page worth of data, internally to the library. This is declared once as a global buffer for safety under low-memory conditions; a page may be too big to use temporary storage on the C stack, and the NAND library deliberately avoids the use of `malloc`.

By convention, a driver package would declare `CYGPKG_IO_NAND` as its parent and use `cyg/devs/nand` as its `include_dir`, but there is no intrinsic reason why this should be so.

High-level (chip) functions

The high-level functions provided by the chip driver are typically created as an *inline file* providing a fully-populated `cyg_nand_dev_fns_v1` struct, instantiated by the `CYG_NAND_FUNS` macro. The high-level driver should not directly read or write to the hardware itself, but instead call into functions in the low-level driver.

The form the low-level functions should take is not prescribed; typically functions will be required to write commands to the device, to read and write data, and to query any status line which may be present. The high-level driver should normally provide a header file containing prototypes for the functions it requires from the low-level

driver. (The low-level source file would provide the low-level functions required, include the high-level include, then instantiate the combined driver using the CYG_NAND_DEVICE macro.)

This source code layout is not intended as a prescription. It would for example be entirely in order to store pointers to the low-level functions in a struct and set *priv* to point to that struct, which could be useful in some cases.

Note: The device driver must not call `malloc` or otherwise allocate memory; all data should be in the stack or set as globals. This is because the driver may be required to run within a minimal eCos configuration.

These functions should all return 0 on success, or a negative eCos error code. In the event of an error, do *not* call back into the NAND library; use the NAND_CHATTER macro to report, in case a human is watching, and return an error code. The library will take care of ensuring the correct response to the application and updating the BBT as necessary.

Device initialisation

```
static int my_devinit (cyg_nand_device *dev);
```

The `devinit` function is the most complex, and logically one to write first. It is responsible for:

- initialising the device, typically by sending a reset command;
- interrogating the device to confirm its presence and properties;
- setting up the partition table list (see "Planning a port" above);
- setting up mutexes as necessary (see "Locking against concurrent access" above);
- populating the other members of the `cyg_nand_device` struct (see below).

Interrogating the device is normally performed by sending a *Read ID* command and examining the result, which typically encodes some or all of the chip parameters.

Given the similarity between many NAND parts, it may be possible to write a generic driver to cover all of one or more manufacturer's parts, or indeed for all ONFI-compliant parts. At the time of writing, this has not yet been attempted.

The `devinit` function must set up the following struct members:

page_bits

The size of the regular (non-spare) part of a page, expressed as the logarithm in base 2 of the number of bytes. For example, if pages are 2048 bytes long, *page_bits* would be 11. Obviously, the size of a page must be an exact power of two.

spare_per_page

The number of bytes of spare area available in each page.

block_page_bits

The base-2 log of the number of pages per eraseblock.

blockcount_bits

The total number of erase blocks in the device, expressed as a base-2 log.

chipsize_log

The total size of the chip, not counting the spare areas. This is required so that the library can double-check that the given parameters make sense by comparing with the preceding fields. Again, this field is itself a base-2 logarithm.

bbt.data

Space for the in-memory Bad Block Table for this device.

bbt.datasize

This is the size of *bbt.data*, in bytes. At present, this should be two bits times the number of blocks in the device; in other words, $1 < (blockcount_bits - 2)$ bytes.

The *cyg_nand_device* struct has two further members *ecc* and *oob* which must be set up to point to the ECC and OOB descriptors to use for the device. This is normally done by the *CYG_NAND_DEVICE* low-level instantiation macro, so will be better described in that section, but at this level you should be aware that it is also safe to set up the descriptor block during *devinit*. For example if multiple semantics might be you had included logic to detect what semantics to use.

The Bad Block Table itself is implemented in a way which intends to be compatible with the Linux MTD layer. A full parameter struct is not currently provided, though one may be in future.

Reading, writing and erasing data

The read and write operations are divided into three phases, with the following flow:

- Begin. This is called once; the driver should lock any platform-level mutex and send the command and address.
- Stride. This is called one or more times to read the page data from the device.

Note: The reason for this is if the platform provides a NAND controller with hardware ECC: it is often necessary to read out the ECC registers every so often.

- Finish. This is called once; it should read or write the spare area, (on programming) send a "program confirm" command and check its status, and unlock any platform-level mutex.

Erasing is a single-shot call which should lock any platform-specific mutex, send the command, check its status and unlock the mutex.

```
static int my_read_begin(cyg_nand_device *dev, cyg_nand_page_addr page);
static int my_read_stride(cyg_nand_device *dev, void * dest, size_t size);
static int my_read_finish(cyg_nand_device *dev, void * spare, size_t spare_size);

static int my_write_begin(cyg_nand_device *dev, cyg_nand_page_addr page);
static int my_write_stride(cyg_nand_device *dev, const void * src, size_t size);
static int my_write_finish(cyg_nand_device *dev, const void * spare, size_t spare_size);
```



```
static int my_erase_block(cyg_nand_device *dev, cyg_nand_block_addr blk);
```

Searching for factory-bad blocks

```
static int my_is_factory_bad(cyg_nand_device *dev, cyg_nand_block_addr blk);
```

The very first time a NAND chip is used, the library has to scan it to check for factory-bad eraseblocks and build up the Bad Block Table. This function is called repeatedly to do so, one block at a time; it should return 1 if the block is marked bad, or 0 if the block appears to be OK.

Typically this function will invoke `read_page`; blocks are usually marked factory-bad by the presence of a particular signature in the out-of-band area of the first or second page of that block.

Warning

It is extremely important that you get this function right; after an eraseblock has been written to, it is no longer possible to reliably determine whether the block was factory-bad. It is never safe to assume that the factory-bad signature for a chip is the same as that of a similarly-sized chip or another by the same manufacturer; *always* check the correct spec sheet for the actual part or part-family in use!

Tip: Because this function is critical and a subtle error could cripple your application some time later in the field when it runs across undetected factory-bad blocks, you might find it handy to have a double-check before proceeding. If you enable `CYGSEM_IO_NAND_READONLY` in your eCos configuration during early development, you can safely fire up a test application (which calls `cyg_nand_lookup`) whilst watching the chatter output: the scan will be performed, but no BBT will be written. You can then compare the number of bad blocks reported against the manufacturer's specification of the maximum. Double-check that your `is_factory_bad` function is correct before enabling read-write mode!

Declaring the function set

```
CYG_NAND_FUNS_V2(mydev_funs, my_devinit,
    my_read_begin, my_read_stride, my_read_finish,
    my_write_begin, my_write_stride, my_write_finish,
    my_erase_block, my_is_factory_bad);
```

This macro ties the above functions together into a struct whose name is given as its first argument. The name of the resulting struct must be quoted when the driver is formally instantiated, which is normally done by the low-level functions.

Note: Earlier versions of this library used a slightly different device interface, keyed off the macro `CYG_NAND_FUNS`. This interface has been retired.

Low-level (board) functions

The set and prototypes of the functions required here will necessarily depend on the board and to a lesser extent on the NAND part itself. The following functionality is typically required:

- Very low-level hardware initialisation - for example, GPIO pin direction and interrupt config - if this has not already been done by the platform HAL
- Set up the chip partition table (see below)
- Runtime hardware config as required, such as commanding an FPGA or CPLD to route lines to the NAND part
- Write a command (byte)
- Write an address (handful of bytes)
- Write data, usually at the chip's full bus width (typically 8 or 16 bits)
- Read data at full bus width
- Read data at 8-bit width (if the chip has a 16 bit data bus, some commands - commonly ReadID - may return 8-bit data)
- Poll any status lines required or - if supported - set them up as interrupts to allow sleeping-wait

Talking to the chip

It is impossible to prescribe how to achieve this, as it depends entirely on how the NAND part is wired up on the board.

The ideal situation is that the NAND part is wired in via the CPU's memory controller and that the controller is set up to do most of the hard work for you. In that case, reading and writing the device is as simple as accessing the correct memory-mapped I/O address; usually different address ranges connect to the device's command, address and data registers respectively.

Tip: The HAL provides a number of macros in `<cyg/hal/hal_io.h>` to read and write memory-mapped I/O.

Note: On platforms with an MMU, MMIO may be rerouted to different addresses to those on the board spec sheet. Check the MMU setup in the platform HAL.

On some platforms, you may have to invoke an FPGA or CPLD to be able to talk to the NAND chip. This might typically take the form of a handful of MMIO accesses, but should hopefully be fairly straightforward once you've figured out how the components interrelate.

The worst case is where you have no support from any sort of controller hardware and have to bit-bang GPIO lines to talk to the chip. This is a much more involved process; you have to take great care to get the timings right with carefully tuned delays. The result is usually quite CPU intensive, and could be clock speed sensitive too; you should check for and take account of any CDL settings in the architecture and variant HAL which allow the CPU clock frequency to be changed.

Tip: If your low-level functions take a `cyg_nand_device` pointer as an argument, you can use its `priv` member to hold or point to some relevant data like the MMIO addresses to use, which is preferable to hard-coding them. Indeed, if you wish your board port to support more than one chip, you should use the `priv` member to distinguish between them.

Setting up the chip partition table

It is the responsibility of the high-level `devinit` function to set up the device's partition table. (It may be appropriate for it to invoke a low-level function to do this.)

The partition definition is an array of `cyg_nand_partition` entries in the `cyg_nand_device`.

```
struct _cyg_nand_partition_t {
    cyg_nand_device *dev;
    cyg_nand_block_addr first;
    cyg_nand_block_addr last;
};
typedef struct _cyg_nand_partition_t cyg_nand_partition;

struct _cyg_nand_device_t {
    ...
    cyg_nand_partition partition[CYGNUM_NAND_MAX_PARTITIONS];
    ...
};
```

Application-visible partition numbers are simply indexes into this array.

- On a live partition, `dev` must point back to the `cyg_nand_device` containing it. If `NULL`, the partition is inactive.
- `first` is the number of the first block of the partition.
- `last` is the number of the last block of the partition (*not* the number of blocks, unless the partition starts at block 0).

Putting it all together...

Finally, with everything else in place, we turn to the `CYG_NAND_DEVICE` macro to instantiate it.

```
CYG_NAND_DEVICE(my_nand, "onboard", &mydev_funs, &my_priv_struct, &linux_mtd_ecc, &nand_mtd_oob_64
```

In order, the arguments to this macro are:

- The name to give the resultant `cyg_nand_device` struct;
- the device identifier string, application-visible to be used in `cyg_nand_lookup()`;
- a pointer to the device high-level function set to use, normally set up by the `CYG_NAND_FUNS` macro;
- the `priv` member to include in the struct;

- a pointer to the ECC semantics block to use. `linux_mtd_ecc` provides software ECC compatible with the Linux MTD layer, but it is strongly recommended to use onboard hardware ecc support if this is present as it gives a huge speed boost. See [the Section called ECC implementation](#) for more details.
- a pointer to the OOB-area layout descriptor to use (see `nand_oob.h`: `nand_mtd_oob_16` and `nand_mtd_oob_64` are Linux-compatible layouts for devices with 16 and 64 bytes of spare area per page respectively).

The macro invokes the appropriate linker magic to pull all the compiled NAND device structs into one section so the NAND library can find them.

ECC implementation

The use of ECC is strongly recommended with NAND flash parts owing to their tendency to occasionally bit-flip. This is usually done with a variant of a Hamming code which calculates column and line parity. The computed ECC is stored in the spare area of the page to which it relates.

The NAND library automatically computes and stores the ECC of data as it is written to the chip. On read, the code is calculated for the data actually read; this is compared with the stored code and the data repaired if necessary.

The NAND library comes with a software ECC implementation named `linux_mtd_ecc`. This is compatible with the ECC used in the Linux MTD layer, hence its name. It calculates a 3-byte ECC on a 256-byte data block. This algorithm is adequate for most circumstances, but it is strongly recommended to use any hardware ECC support which may be available because of the performance gains it yields. (In testing, we observed that up to two thirds of the time taken by every page read and program call was used in computing ECC in software.)

The ECC interface

This library draws a semantic distinction between *hardware* and *software* ECC implementations.

- A software ECC implementation will typically not require an initialisation step. The calculation function will always be called with a pointer to the data bytes to compute.
- A hardware implementation is assumed to read and act upon the data *as it goes past*. Therefore, it will not be passed a pointer to the data when its `calculate` step is invoked.

An ECC is defined by the following parameters:

- The size of data block it handles, in bytes.
- The size of ECC it calculates on those blocks, in bytes.
- Whether the algorithm is hardware or software.

An ECC algorithm must provide the following functions:

```
/* Initialises an ECC computation. May be NULL if not required. */
void my_ecc_init(struct _cyg_nand_device_t *dev);

/* Returns the ECC for the given data block.
```

```

* If IS_HARDWARE:
*   - dat and nbytes are ignored
* If ! IS_HARDWARE:
*   - dat and nbytes are required
*   - if nbytes is less than the chunk size, the remainder are
*     assumed to be 0xff.
*/
void my_ecc_calc(struct _cyg_nand_device_t *dev,
                const CYG_BYTE *dat, size_t nbytes, CYG_BYTE *ecc);

/* Repairs the ECC for the given data block, if needed.
* Call this if your read-from-chip ECC doesn't match what you computed
* over the data block. Both *dat and *ecc_read may be corrected.
*
* 'nbytes' is the number of bytes we're interested in; if a correction
* is indicated outside of that range, it will be ignored.
*
* Returns:
*   0 for no errors
*   1 for a corrected single bit error in the data
*   2 for a corrected single bit error in the ECC
*  -1 for an uncorrectable error (more than one bit)
*/
int my_ecc_repair(struct _cyg_nand_device_t *dev,
                  CYG_BYTE *dat, size_t nbytes,
                  CYG_BYTE *ecc_read, const CYG_BYTE *ecc_calc);

```

The algorithm parameters and functions are then tied together with one of the following macros:

```

CYG_NAND_ECC_ALG_SW(my_ecc, _datasize, _eccsize, my_ecc_init, my_ecc_calc, my_ecc_repair);

CYG_NAND_ECC_ALG_HW(my_ecc, _datasize, _eccsize, my_ecc_init, my_ecc_calc, my_ecc_repair);

```

Tip: It's OK to use software ECC while getting things going, but if you do then switch to a hardware implementation, you probably need to erase your entire NAND chip including its Bad Block Table. The `nanderase` utility may come in handy for this.)

Warning

You must be sure that your ECC repair algorithm is correct. This can be quite tricky to test. However, it is often possible to hoodwink the controller into computing ECCs for you even if the data is not going to affect the data stored on the NAND chip, for example if you send it data but haven't told it to program a page. A variant of the `sweccwalk` test may come in handy for this purpose.

An example implementation, including an ECC calculation and repair test named `eccwalk`, may be found in the STM3210E evaluation board platform HAL, `packages/hal/cortexm/stm32/stm3210e_eval`. The chip NAND controller has on-board ECC calculation, but does not undertake to repair data; a repair function was written specially.

Chapter 9. Tests and utilities

Unit and functional tests

The NAND library includes a number of tests. The most useful to driver writers are `readwrite`, `rwbenchmark` and `sweccwalk`; the others are only likely to be of interest to library maintainers.

readwrite

Performs a read-write-erase cycle on the first NAND device it finds, checking that its operations have had the expected effect on the device contents. This is a potentially destructive test; do not run it on a device containing data you care about!

rwbenchmark

A more involved version of `readwrite`, this is a timing test which performs multiple reads, writes and erases and applies statistical techniques to the results in the same way that `tm_basic` instruments the speed of various eCos kernel functions. *This is a potentially destructive test; do not run it on a device containing data you care about!*

sweccwalk

Repeatedly makes single-bit changes to a data buffer and checks that the software ECC implementation correctly repairs them.

Tip: This test can be adapted to test out hardware ECC implementations. The test outputs the raw ECC codes as it goes, which is useful in confirming that the bits in the computed ECC are what you think they are.

nandunit

Some unit tests which do not require any NAND device: ECC known answer vectors, and OOB area packing/unpacking correctness.

readlimits

Attempts to read a block outside of a partition, confirming that it doesn't work.

There are some further tests of the library which require the [synthetic NAND device](#).

Ancillary NAND utilities

The following utilities are included with the NAND library. They are standalone eCos applications; for convenience, you can set `CYGBLD_IO_NAND_BUILD_UTILS` in your eCos configuration and they will be built and placed into `install/tests/io/nand/current/utils`.

erasenand.c

Loops over all the blocks of a partition, erasing all the blocks which are not marked as bad. The device and partition to erase are set by `#define`.

Note: This will not normally erase the Bad Block Table. This is because the BBT reports its own blocks as "Reserved" when queried via `cyg_nand_bbt_query`, which makes them inaccessible to applications. However, if `CYGSEM_IO_NAND_USE_BBT` is turned off, then any BBT present will not be detected and hence will be erased.

erase_bbt_dangerous.c

Erases the NAND blocks comprising the primary and mirror bad-block tables of a device. The device to erase is set by `#define`. (The tables are detected by the library in the usual way. If none are present, the library will scan the device for factory-bad blocks to create such a table, then this code will immediately erase it.)

Warning

It is particularly dangerous to run this utility on a production device, as it is generally not possible to later reconstruct the list of factory-bad blocks. It is intended only as an aid to driver authors.

Chapter 10. Samsung K9 family NAND chips

Overview

The `CYGPKG_DEVS_NAND_SAMSUNG_K9` driver package currently provides support for the Samsung K9F1G08x0x series of NAND flash chips, and is intended to be expanded to provide support for more of the K9 family.

Most users will only need to add this package to their eCos configuration and not need to interact with it further. This package provides only an inline code fragment which is intended to be instantiated by the target platform HAL and provided with appropriate board-specific low-level functions allowing it to access the hardware.

Note: This part is not quite ONFI-compliant, but this code could probably be extended to a much wider set of chips - or indeed to the ONFI specification - without too much trouble. Appropriate definitions will be required for the chip identifier, decoding of the Read ID response, and the chip's blockcount-bits and device-size fields.

Note: At the present time, this driver has the limitation that it only supports 8-bit parts. This is an area of probable future expansion.

Using this driver in a board port

This driver's chip support is currently provided as two files:

`cyg/devs/nand/k9fxx08x0x.h`

Prototypes the low-level chip access functions required by the chip driver and declares a private struct for use by the driver.

`cyg/devs/nand/k9fxx08x0x.inl`

Implements high-level chip functions and exposes them via the `CYG_NAND_FUNS` macro. This file is not intended to be compiled on its own.

A platform HAL would typically make use of this driver in a single source file with the following steps:

- Declare a private struct and one or more static instances of it as appropriate,

- `#include <cyg/devs/nand/k9fxx08x0x.h>`
- implement the required low-level functions,
- `#include <cyg/devs/nand/k9fxx08x0x.inl>`
- finally, instantiate the chip with the `CYG_NAND_DEVICE` macro the appropriate number of times, giving each chip an appropriate name, its own private struct if need be, and selecting the ECC and OOB semantics to use.

For more details about the infrastructure provided by the NAND layer and the semantics it expects of the chip driver, refer to [Part VII in *eCosPro® Reference Manual*](#). An example driver instantiation can be found in [the NAND driver for the EA LPC2468 platform](#).

Memory usage

As discussed in [the Section called *High-level \(chip\) functions* in Chapter 8](#), the chip initialisation function must set up the `bbt.data` pointer in the `cyg_nand_device` struct. This driver does so by including a sufficiently large byte array in the `k9_priv` struct. That struct is intended to be allocated as a static struct in the data or BSS segment (one per chip), which avoids adding a dependency on `malloc`.

Low-level functions required from the platform HAL

These functions are prototyped in `k9fxx08x0x.h`. They have no return value ("void"), except for `read_data_1` which returns the byte it has read.

`write_cmd(device, command)`

Writes a single command byte to the chip's command latch.

`write_addrbytes(device, pointer to bytes, number of bytes)`

Writes a number of address bytes in turn to the chip's address latch.

`CYG_BYTE read_data_1(device), read_data_bulk(device, output pointer, number of bytes)`

Reads data from the device, respectively a single byte and in bulk.

`write_data_1(device, byte), write_data_bulk(device, data pointer, number of bytes)`

Writes data to the device, respectively a single byte and in bulk.

`wait_ready_or_time(device, initial delay, fallback time)`

Wait for the chip to signal READY line or, if this line is not available, fall back to a worst-case time delay (measured in microseconds).

wait_ready_or_status(device, mask)

Wait for the chip to signal READY line or, if this line is not available, enter a loop waiting for its Status register (ANDed with the given mask) to be non-zero.

k9_devlock(device), k9_devunlock(device)

Hooks for any board-specific locking which may be required in addition to the NAND library's chip-level locking. (This would be useful if, for example, access to multiple chips was mediated by a single set of GPIO lines which ought not to be invoked concurrently.)

k9_plf_init(device)

Board-level platform initialisation hook. This is called very early on in the chip initialisation routine; it should set up any locking required by the devlock and devunlock functions, interrupts for the driver and any further lines required to access the chip as appropriate. *Once this has returned, the chip driver assumes that the platform is fully prepared for it to call the other chip access functions.*

k9_plf_partition_setup(device)

Board-level partition initialisation hook. This should set up the `partition` array of the device struct in a way which is appropriate to the platform. For example, the partitions may be set as fixed ranges of blocks, or by CDL. This is called at the end of the chip initialisation routine and may, for example, call into the chip to read out a "partition table" if one is present on the board. *If you do not set up partitions, applications will not be able to use the high-level chip access functions provided the NAND library.*

Chapter 11. ST Microelectronics NANDxxxx3a chips

Overview

The `CYGPKG_DEVS_NAND_ST_NANDXXXX3A` driver package provides support for the NANDxxxx3A chip family by ST Microelectronics.

Most users will only need to add this package to their eCos configuration and not need to interact with it further. This package provides only an inline code fragment which is intended to be instantiated by the target platform HAL and provided with appropriate board-specific low-level functions allowing it to access the hardware.

Using this driver in a board port

This driver's chip support is currently provided as two files:

`cyg/devs/nand/nandxxxx3a.h`

Prototypes the low-level chip access functions required by the chip driver, declares a private struct for use by the driver and provides a `NANDXXXX3A_DEVICE` macro for convenience.

`cyg/devs/nand/nandxxxx3a.inl`

Implements high-level chip functions and exposes them via the `CYG_NAND_FUNS` macro. This file is not intended to be compiled on its own.

A platform HAL would typically make use of this driver in a single source file with the following steps:

- Declare a private struct and one or more static instances of it as appropriate,
- `#include <cyg/devs/nand/nandxxxx3a.h>`
- implement the required low-level functions,
- `#include <cyg/devs/nand/nandxxxx3a.inl>`
- finally, instantiate the chip with the `NANDXXXX3A_DEVICE` macro the appropriate number of times, giving each chip an appropriate name and its own private struct if need be, declaring its size, and selecting the ECC and OOB semantics to use.

For more details about the infrastructure provided by the NAND layer and the semantics it expects of the chip driver, refer to [Part VII in eCosPro® Reference Manual](#). An example driver instantiation can be found in the platform HAL for the STM3210E-EVAL board.

Memory usage note

As discussed in [the Section called High-level \(chip\) functions in Chapter 8](#), the chip initialisation function must set up the `bbt.data` pointer in the `cyg_nand_device` struct. This driver does so by including pointer to a sufficiently large byte array in the `nandxxx3a_priv` struct. That struct is intended to be allocated as a static struct in the data or BSS segment (one per chip), which avoids adding a dependency on `malloc`.

Low-level functions required from the platform HAL

These functions are prototyped in `nandxxxx3a.h`. They have no return value ("void"), except where indicated.

`write_cmd(device, command)`

Writes a single command byte to the chip's command latch.

`write_addrbytes(device, pointer to bytes, number of bytes)`

Writes a number of address bytes in turn to the chip's address latch.

CYG_BYTE `read_data_1(device), read_data_bulk(device, output pointer, number of bytes)`

Reads data from the device, respectively a single byte and in bulk.

`write_data_1(device, byte), write_data_bulk(device, data pointer, number of bytes)`

Writes data to the device, respectively a single byte and in bulk.

`wait_ready_or_time(device, initial delay, fallback time)`

Wait for the chip to signal READY or, if this line is not available, fall back to a worst-case time delay (measured in microseconds).

`wait_ready_or_status(device, mask)`

Wait for the chip to signal READY or, if this line is not available, enter a loop waiting for its Status register (ANDed with the given mask) to be non-zero.

nandxxx3a_devlock(device), nandxxx3a_devunlock(device)

Hooks for any board-specific locking which may be required in addition to the NAND library's chip-level locking. (This would be useful if, for example, access to multiple chips was mediated by a single set of GPIO lines which ought not to be invoked concurrently.)

int nandxxx3a_plf_init(device)

Board-level platform initialisation hook. This is called very early on in the chip initialisation routine; it should set up any locking required by the devlock and devunlock functions, interrupts for the driver and any further lines required to access the chip as appropriate. *Once this has returned, the chip driver assumes that the platform is fully prepared for it to call the other chip access functions.*

int nandxxx3a_plf_partition_setup(device)

Board-level partition initialisation hook. This should set up the `partition` array of the device struct in a way which is appropriate to the platform. For example, the partitions may be set as fixed ranges of blocks, or by CDL. This is called at the end of the chip initialisation routine and may, for example, call into the chip to read out a "partition table" if one is present on the board. *If you do not set up partitions, applications will not be able to use the high-level chip access functions provided the NAND library.*

VIII. Synthetic Target NAND Flash Device

Synthetic Target NAND Flash Device

Name

Synthetic Target NAND Flash Device — Emulate NAND flash hardware in the synthetic target

Overview

The device driver `CYGPKG_DEVS_NAND_SYNTH` emulates NAND flash hardware inside the eCos synthetic target. In addition it provides a number of debug facilities which cannot readily be implemented on real embedded hardware, including:

1. The emulated NAND contents are held on a file in the Linux host. This makes it easy to archive and restore NAND images, allowing test runs to be repeated with the exact same state each time.
2. The device driver can log details of all NAND I/O to a separate logfile in the Linux host. This makes it easier to work out exactly what is happening in the application, and more importantly it can help with figuring out what went wrong when. For extended runs it is possible to limit the disk space used for logging. It is also possible to generate checkpoints, where the current NAND image is saved to a separate file.
3. It is possible to inject bad blocks at run-time, to check how the application would cope on real hardware if and when a NAND erase block developed a fault. These can be made to affect random blocks or specific blocks, for example ones holding filesystem metadata.

Some of the functionality is always available and uses compile-time configuration via CDL. This allows applications to be run stand-alone. The more advanced functionality such as logging and bad block injection is only available when running in conjunction with the synthetic target I/O auxiliary, when `--io` is used on the command line. The settings for logging and bad block injection usually come from the `default.tdf` target definition file. These can be changed on a per-run basis by adding `--nanddebug` to the command line, which will cause a suitable dialog box to pop up during NAND driver initialization.

Compile-time Configuration

This package `CYGPKG_DEVS_NAND_SYNTH` will automatically be loaded when creating a new eCos configuration for the Linux synthetic target. However the package will be inactive until the generic NAND support is added to the configuration.

The synthetic target NAND driver has been designed to be functional both when running stand-alone and when used with the I/O auxiliary. Hence some of the basic parameters of the emulated NAND device must be specified at compile-time, and this is handled via CDL configuration options.

`CYGDAT_NAND_SYNTH_FILENAME` specifies the host-side file that will be used to hold the NAND data. The default is `synth_nand.dat` in the current directory. If the file does not exist then the driver will create it during initialization. All data in a newly-created image file will be set to `0xFF`, corresponding to an erased device. Hence deleting the current image file makes it possible to start a test run with a blank NAND device.

A NAND device consists of some number of erase blocks: erase operations affect all data in an erase block. Erase blocks are made up of some number of pages, and write operations typically affect a page at a time. Each page consists of a main data block plus some spare bytes, also known as out of

band or OOB data. There are four CDL configuration options controlling the size and layout of the emulated flash device: `CYGNUM_NAND_SYNTH_BLOCK_COUNT`, `CYGNUM_NAND_SYNTH_PAGES_PER_BLOCK`, `CYGNUM_NAND_SYNTH_PAGESIZE`, and `CYGNUM_NAND_SYNTH_SPARE_PER_PAGE`. The default settings are 1024 erase blocks, 32 pages per block, 2K of data per page, and 64 bytes of OOB data per page. This gives an emulated device size of 64MB plus 2M OOB.

The size and layout parameters are encoded in each NAND image file. If these configuration options are changed then existing image files will be incompatible and the device driver will report a fatal error at run-time. This avoids compatibility problems with higher-level code: if a file system has formatted the NAND device for a 2K page size then it is likely to get very confused if the page size suddenly changes to 512 bytes.

The NAND device can be partitioned manually by enabling the component `CYGSEM_DEVS_NAND_SYNTH_PARTITION_MANUAL_CONFIG` and manipulating the options below this. The default is for a single partition occupying the entire NAND device.

Option `CYGSEM_NAND_SYNTH_RANDOMLY_LOSE` activates code in the driver which triggers frequent bit errors during read operations. These should be handled by error correcting codes within the generic NAND layer so should be transparent to higher-level code. The option exists mainly as an easy way of testing the automatic error correction support.

Run-time Customization

Logging and bad block injection are controlled by run-time customization via the synthetic target I/O auxiliary, not by compile-time CDL options. This allows the same test executable to be run with and without logging or with different sequences of injected bad blocks. If the executable is run without the I/O auxiliary, without the `--io` command line option, then both logging and bad block injection will be disabled.

The main way of customizing both logging and bad block injection is via the target definition file, usually `default.tdf`. The driver comes with a file `nand.tdf` holding the various options and explanatory text. This file should be incorporated into `default.tdf` and edited as appropriate. Note that all NAND-related settings should be inside a `synth_device nand` section.

Logging

The target definition file settings related to logging are as follows:

```
logfile           "/tmp/synthnand.log"
log               read write erase error
max_logfile_size  16M
number_of_logfiles 4
generate_checkpoint_images
```

The `logfile` setting controls the location of the logfile. The default is to add a suffix `.log` to the `CYGDAT_NAND_SYNTH_FILENAME` setting, thus creating logfiles in the same directory as the NAND image file.

The `log` setting specifies which events should be logged. It is followed by a list of some or all of the following: `read`, `READ`, `write`, `WRITE`, `erase`, and `error`. `read` logs all calls to the driver's page read function, but not the data actually read. `READ` is like `read` but logs the actual data as well as the event. Similarly `write` and `WRITE` log calls to the driver's page write function, without or with the data being written. `erase` logs calls to the driver's block erase function. `error` logs any bad block injection events.

Logging can quickly generate very large files, especially when `READ` or `WRITE` debugging is enabled. This can have unfortunate side effects, for example an overnight stress test can fail because the logfile has filled all available disk space. To avoid this it is possible to limit the size of each logfile using the `max_logfile_size` setting. This is simply a number followed by a unit K, M or G.

When `max_logfile_size` is exceeded the NAND driver takes appropriate action. The default behaviour is just to delete the current logfile and create a new one, with the same name as before. This can be unfortunate if some particularly interesting event happened just before the maximum logfile size was exceeded because all logging information related to that event will have been lost. To avoid this it is possible to have multiple logfiles, limited by the `number_of_logfiles` setting. Assume a logfile name of `/tmp/synthnand.log`, a maximum logfile size of 16M and four logfiles. After the first 16MB of logging data has been written to `/tmp/synthnand.log` that file will be renamed to `/tmp/synthnand.log.0` and a new current logfile `/tmp/synthnand.log` will be created. After another 16MB the current logfile will be renamed to `/tmp/synthnand.log.1`, and so on. After 64MB the maximum allowed number of logfiles has been created, so `/tmp/synthnand.log.0` will be deleted and then `/tmp/synthnand.log` will be renamed to `/tmp/synthnand.log.3`. Hence the maximum disk space used will be between 48MB and 64MB, plus a small amount of overflow for each logfile. All logfiles are in [plain text](#), one event per line, and a single event will not be spread over multiple logfiles.

In addition to the logfiles the NAND driver will generate image checkpoint files if `generate_checkpoint_images` is enabled. At the start of the test run the driver will copy the current NAND image to a new file `/tmp/synthnand.log.checkpoint`, using the same base filename as for logfiles. If support for multiple logfiles is enabled then the current checkpoint file will be renamed in the same way and at the same time as the current logfile, and a new checkpoint file will be created using the current image data. Hence for any logfile it is possible to examine both the starting image and the final image (which may be the current one).

Bad Block Injection

There are two settings related to bad block injection: `factory_bad` and `inject`. The former is straightforward:

```
factory_bad 17 42 256 1019
```

This setting is used only when the NAND image file does not yet exist and the driver has to create a new one. The specified erase blocks are marked as factory-bad and hence unusable. The setting consists simply of one or more numbers, each in the range 0 to `CYGNUM_NAND_SYNTH_BLOCK_COUNT-1`. A maximum of 32 blocks can be marked factory bad.

Bad block injection is rather more complicated, in an attempt to make it sufficiently flexible for a variety of uses. The target definition file can contain one or more `inject` settings. There is a limit of eight erase definitions and eight write definitions, giving a maximum of sixteen bad block injection definitions. However some of these can use `repeat`, so the number of bad blocks injected during a test run is limited only by the size of the emulated device. Example settings are:

```
inject erase current after rand% 1024 erases
inject write current after rand% 100000 calls repeat
inject erase block 1 after 3 block_erases
inject write page 9860 after 1000 writes
```

The `inject` keyword should be followed by either `erase` or `write`. If `erase` then the bad block injection happens during a call to the driver's erase function, otherwise it happens during a call to `write`. This is followed by a block or page definition, the keyword `after`, an event counter, and a couple of optional flags.

The simplest block or page definition is the keyword `current`. This simply means that whichever block is being erased, or whichever block contains the page being written, will be marked bad. Typically this will be used for injecting random faults. If a given block is the subject of an above-average number of erase or write operations then it is more likely to be the current block in one of these definitions, so heavily-used blocks are more likely to fail.

The alternative to `current` is to list a specific block for an erase definition, or a specific page for a write definition. Blocks go from 0 to `CYGNUM_NAND_SYNTH_BLOCK_COUNT - 1`. Pages go from 0 to $(\text{CYGNUM_NAND_SYNTH_BLOCK_COUNT} * \text{CYGNUM_NAND_SYNTH_PAGES_PER_BLOCK}) - 1$. This can be particularly useful when testing higher-level code that uses certain blocks specially, for example for storing filesystem metadata. It can also be useful when attempting to repeat a test run using information from a logfile.

The fields immediately following the `after` keyword specify when the bad block injection should trigger. They consist of the optional keyword `rand%`, a count, and an event identifier. If `rand%` is not specified then exactly the specified number of events must occur before the bad block injection triggers. Otherwise some number between 0 and count-1 events must occur. The event identifier can be one of `erases`, `writes`, `calls`, `block_erases`, or `page_writes`. `erases` means the number of calls to the driver's block erase function. `writes` means the number of calls to the page write function. `calls` means the total number of read, write or erase calls. `block_erases` means erase calls for a specific block. Similarly `page_writes` means write calls for a specific page. `block_erases` and `page_writes` cannot be used together with `current`, only with a specific block or page.

So, consider the first example again:

```
inject erase current after rand% 1024 erases
```

The driver will calculate a random number between 0 and 1023, say 427. Once there have been at least 427 erase calls the bad block injection will trigger. Since the definition is for `erase current`, the injection can happen immediately. Hence whichever block is specified during the 427th erase call will be marked bad and that erase call will fail with error code `EIO`.

Suppose that the definition used `write current` instead of `erase current`. The definition will still trigger during erase call 427, but it will not take effect immediately. Instead whichever page is the subject of the next write operation will be marked bad. Alternatively, suppose that the definition was for `erase block 42` but call 427 was for block 512 instead. If some time after call 427 there was another erase call for block 42, that later erase call will fail and cause the block to be marked bad.

Now consider the next definition:

```
inject write current after rand% 100000 calls repeat
```

This event will trigger after between 0 and 99999 calls into the driver. These calls can be reads, writes, or erases. If the triggering call is a write then the block affected will be the one containing the page being written. Otherwise whichever page is the subject of the next write operation will be the one affected.

```
inject erase block 1 after 3 block_erases
```

This definition will only ever affect block 1. The first two calls to erase block 1 will succeed. The third call will fail. Erase calls for any other block have no effect on this definition.

```
inject write page 9860 after 1000 writes
```

This definition will only ever affect page 9860. The definition will trigger after 1000 writes to any page. If the thousandth write happens to be for page 9860, it will fail. Otherwise the next write for page 9860 will fail, whenever that happens. If the definition specified event type `page_writes` instead of `writes` it would trigger only after 1000 writes to page 9860 instead of 1000 writes to any page.

Trigger definitions can be followed by two optional keywords. The first is `repeat` and indicates that the definition should trigger multiple times, not just once. `repeat` can only be used for block or page `current`, not for a specific block or page, since any given block can only fail once. The second keyword is `disabled`. This can be used to create a bad block injection definition which is not active by default but which can be enabled in the GUI interface.

All bad block injection definitions operate in parallel, not sequentially. It is possible for multiple definitions to trigger during a single call but a given block can only fail once.

Interactive Dialog

Although it is possible to change the logging and other settings between test runs by editing the target definition file, the package also provides a way of changing these during driver initialization time. If the command line option `--nanddebug` is used in addition to `--io` then the I/O auxiliary will pop up a dialog box allowing the various settings to be edited.

Files		Logging	Errors
NAND image file: synth_nand.dat Page size: 2048 Spare (00B) size: 64 Pages per erase block: 32 Number of erase blocks: 1024			
Delete	synth_nand.dat		
Save	synth_nand.dat.bak		
Save As ...			
Restore	synth_nand.dat.bak		
Restore from ...			
Refresh			
Checking for existing image file... yes Checking for existing archive file... no			
<input type="button" value="OK"/>			

The dialog consists of a tabbed notebook with separate tabs for Files, Logging, and Errors. The default tab is Files. This shows the NAND device settings as configured via CDL options. If a current NAND image file exists then it can be deleted with a single click, allowing the test run to proceed with a new blank NAND device. The current image file can be saved away to an archive, either using a default name based on the current image name or by letting the user select a file. It is also possible to restore a previously-saved archive, again using the default name if that archive exists, or by selecting an alternative file. If any of the relevant files are changed outside this dialog then the refresh button forces the dialog to check the filesystem again. The various operations will be reported at the bottom of the dialog.

If the Logging tab is selected then the dialog changes to the following:

Files	Logging	Errors
Events to Log <input checked="" type="checkbox"/> read <input type="checkbox"/> READ <input checked="" type="checkbox"/> write <input type="checkbox"/> WRITE <input checked="" type="checkbox"/> erase <input checked="" type="checkbox"/> error		
Logfile settings Logfile: /tmp/synthnand.log <input type="text" value="..."/> Maximum logfile size <input type="text" value="16"/> <input type="text" value="M"/> Number of logfiles <input type="text" value="4"/> Generate checkpoint files <input type="checkbox"/>		
Checking for existing image file... yes Checking for existing archive file... no		
<input type="button" value="OK"/>		

There are separate checkboxes for the six types of events that can be logged. It is also possible to change the current logfile and to edit the other settings related to logfiles.

The Errors tab is not yet implemented.

The dialog can be dismissed using the OK button or by hitting the ESC key. At that point the eCos application will resume running with the selected settings.

File Formats

A NAND image is a binary file with six sections: a 64-byte header block; an array of erase counts; an array of write counts; an array of the factory-bad blocks; a bitmap of the good and bad blocks; and the actual data. The header consists of the following data structure:

```
struct header {
    cyg_uint32 magic;           // 0xEC05A11F
```

```

    cyg_uint32 page_size;          // CYGNUM_DEVS_NAND_SYNTH_PAGESIZE
    cyg_uint32 spare_size;        // CYGNUM_DEVS_NAND_SYNTH_SPARE_PER_PAGE
    cyg_uint32 pages_per_block;   // CYGNUM_DEVS_NAND_SYNTH_PAGES_PER_BLOCK
    cyg_uint32 number_of_blocks;  // CYGNUM_DEVS_NAND_SYNTH_BLOCK_COUNT
    cyg_uint32 tv_sec;
    cyg_uint32 tv_usec;
    cyg_uint32 spare[9];
};

```

All integers in the header and in the following three sections are stored in bigendian format. The *magic* field is used to check that a file really holds a NAND image. The next four fields specify the emulated device size and layout, as per the corresponding CDL options. The *tv_sec* and *tv_usec* fields are filled in with the result of a `gettimeofday()` system call during driver initialization. These fields also appear in logfiles, so code can check that a logfile and an image file correspond to the same test run.

Following the header is an array of `BLOCK_COUNT` integers holding the number of erase calls for each block. Next is an array of `(BLOCK_COUNT * PAGES_PER_BLOCK)` integers holding the number of write calls for each page. These arrays can be used to check that higher-level code is performing wear levelling.

The array of factory-bad blocks consists of 32 integers holding the settings of the target definition's file `factory_bad` setting at the time that the image file was created.

The bitmap following the factory-bad block array holds the current state of each erase block. Bit 0 of byte 0 corresponds to erase block 0, bit 0 of byte 1 corresponds to erase block 8, and so on. If a bit is set then the erase block is ok. If a bit is clear then the erase block is bad, either factory-bad or because it has failed subsequently due to a bad block injection.

The bad block bitmap is followed by the actual data. This consists of all erase blocks concatenated without padding, starting with erase block 0. Each erase block is stored starting from page 0 within that block, and again all the pages are concatenated without padding. For each page the actual data is stored first, followed by the spare (OOB) data.

A logfile is a plain text file, not a binary file. It holds one log event per line. Some of these lines can be rather long if `READ` or `WRITE` logging is enabled. The fields within each line are separated by a single space. The first field indicates the type of record. The next two fields are call counts, one for the event in question and one for the total number of calls into the driver. The remaining fields depend on the record type.

```
I 0 0 1247514233 562000 synth_nand.dat 2048 64 32 1024
```

This is an initialization record when the logfile is created. There have been no calls to the driver yet so the two counts are 0. The next two fields are the *tv_sec* and *tv_usec* timestamp values which are also written into the NAND image file. This allows logfiles and image files to be matched up. The remaining fields identify the page size, the spare size, the number of pages per erase block, and the number of erase blocks.

```
F 2 4 43 0
```

This is a call into the driver's `factorybad` function. It is the second such call, and the fourth call into the driver. The query is for erase block 42, and that block has not been marked as factory-bad.

```
r 5 12 32736 0x0200f0c0 2048 0x0200eeb0 64
```

This is a `read` event. It is the fifth page read into the driver and the 12th call. The request is for page number 32736. 2048 bytes of data should be read into a buffer at location 0x0200f0c0, and 64 bytes of OOB data should be read into 0x0200eeb0.

```
Rd 5 12 32736 0x0200f0c0 2048 FFFFFFFFFF...
Ro 5 12 32736 0x0200eeb0 64 FFFFFFFFFF...
```

These are two `READ` log events corresponding to the previous `read`. The first line `Rd` is for the data part, and the final field consists of 4096 bytes of hexadecimal data. The second line `Ro` is for the OOB part and the final field consists of 128 bytes of hexadecimal data.

```
w 1 1030 32736 0x0200f0c0 2048 0x0200eed0 64
Wd 1 1030 32736 0x0200f0c0 2048 FFFFFFFFFF3FFFFFFFFFCF...
Wo 1 1030 32736 0x0200eed0 64 FFFFFFFFFF426274...
```

These lines show a `write` log event and a `WRITE` log event for the same call into the driver. The fields are the same as for `read` and `READ`.

```
E 2 1031 1022
```

This logs an erase call into the library for block 1022. It is the second erase call, and by this time there have been 1031 calls into the driver.

```
Bb 1 2857 631
...
Bp 2 4012 3189 99
```

These lines show bad block injections. The first is for an erase operation for block 631. That erase call is about to fail with `EIO` and the block will be marked bad. The second is for a write operation for page 3189, which is part of erase block 99. That write operation is about to fail and all of erase block 99 will be marked bad.

Installation

Before a synthetic target eCos application can use a NAND device it is necessary to build and install host-side support. The relevant code resides in the `host` subdirectory of the synthetic target NAND package and building it involves the standard **configure**, **make** and **make install** steps. The implementation of the NAND support does not require any executables, just a Tcl script `nand.tcl` and some support files, so the **make** step is a no-op.

There are two main ways of building the host-side software. It is possible to build both the generic host-side software and all package-specific host-side software, including the NAND support, in a single build tree. This involves using the **configure** script at the toplevel of the eCos repository. For more information on this, see the `README.host` file at the top of the repository. Note that if you have an existing build tree which does not include the synthetic target NAND support then it will be necessary to rerun the toplevel configure script: the search for appropriate packages happens at configure time.

The alternative is to build just the host-side for this package. This requires a separate build directory, building directly in the source tree is disallowed. The **configure** options are much the same as for a build from the toplevel,

and the `README.host` file can be consulted for more details. It is essential that the NAND support be configured with the same `--prefix` option as other eCos host-side software, especially the I/O auxiliary provided by the synthetic target architectural HAL package, otherwise the I/O auxiliary will be unable to locate the NAND support.

Test programs

bbt

Bad Block Table unit test. Finds a readable block, then fiddles with its status in the BBT confirming expected behaviour. Requires the synthetic NAND device.

multipagebbt

As for *bbt* but insists that the device parameters mean that the BBT spans multiple pages on-chip. (This is perhaps a contrived case, but might crop up in future with larger devices, so needed to be tested.)

eccdamage

An ECC error fuzzing exercise. Requires `CYGSEM_NAND_SYNTH_RANDOMLY_LOSE`, which induces pseudo-random bit errors; after 1,000 runs, the number of errors corrected is reported.

IX. NXP LPC2xxx variant HAL

Overview

Name

eCos Support for the NXP LPC2xxx ARM microcontrollers — Overview

Description

The NXP LPC2xxx series of ARM microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip serial, watchdog, RTC (wallclock) and Flash devices. In addition it provides common functionality and definitions that LPC2xxx based platform ports may require, as well as definitions useful to application developers.

This documentation covers the LPC2xxx functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the LPC2xxx variant HAL provides. The areas that are specific to platform HALs and not the LPC2xxx variant HAL include:

- memory map and related configuration and setup
- memory remapping
- External Memory Controller (EMC) and Memory Accelerator Module (MAM) setup (if applicable)
- Definitions of clock (OSC) inputs to PLL
- PLL setup
- PINSEL and GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED control

This variant HAL provides helper macros and defines for some of these, but their use or otherwise is governed by the platform HAL.

On-chip subsystems and peripherals

Name

On-chip subsystems and peripherals — Hardware support

Hardware support

On-chip memory

The NXP LPC2xxx parts include a small amount of on-chip SRAM, and a limited amount of on-chip Flash. Specific capacities vary between parts. The SRAM is generally too small to be useful to RedBoot and some form of external RAM is employed if remote debugging is required, although some limited applications can be run if a GDB stub ROM image is programmed to internal Flash instead. Otherwise for processor models with no external RAM, applications must be programmed directly to internal Flash. The platform HAL may opt to use the SRAM to store the interrupt vectors mapped to address 0, or as a buffer for reprogramming internal flash when using the LPC2xxx Flash driver. The on-chip Flash is generally sufficient to include RedBoot or a GDB stub ROM image, although the on-chip SRAM may not be - again consult the platform HAL documentation. At this time, there is no support for initial programming of the on-chip Flash and so the NXP LPC2000 Flash Utility, Flash Magic or a JTAG/ICE is generally used for this.

Typically, an eCos platform HAL port will expect a RedBoot image to be programmed into the LPC2xxx on-chip Flash memory for development, and the board would boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels.

Serial I/O

The LPC2xxx variant HAL supports basic polled HAL diagnostic I/O over either of the two on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for both on-chip serial devices. The serial driver consists of two eCos packages: `CYGPKG_IO_SERIAL_GENERIC_16X5X` which is a “generic” package for 16x5x compatible serial devices; and `CYGPKG_IO_SERIAL_ARM_LPC2XXX` which provides more specific definitions for the LPC2xxx on-chip serial devices. Using the HAL diagnostic I/O support, either of these devices can be used by RedBoot for communication with the host. If you are only using UART 0, a small amount of memory can be saved by reducing the number of communication channels with the CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS` from 2 to 1 if the platform HAL permits this. It is not possible to enable UART 1 but not UART 0 at this time. If a serial device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication with the HAL I/O support. The alternative serial port should be used instead, if available on the platform. The serial driver supports the line status and modem control (including hardware handshaking) lines on UART 1 only.

Watchdog

A device driver is included for the on-chip watchdog device. This driver allows the use of the standard eCos watchdog API accessible with the `CYGPKG_IO_WATCHDOG` eCos package. If the watchdog is not reset within a time

period defined in the watchdog device driver CDL, an interrupt is generated and a user-supplied function called. Alternatively it may be configured to automatically reset the system.

The watchdog device is also used to implement reset functionality, such as required by the RedBoot **reset** command. It may also be called directly by applications using the following function:

```
#include <cyg/hal/hal_diag.h>
extern void hal_lpc2xxx_reset_cpu(void);
```

RTC/Wallclock

Support is provided for the on-chip RTC (wallclock) device. This allows the use of the standard eCos wallclock API accessible with the `CYGPKG_IO_WALLCLOCK` eCos package. The wallclock is also used by other eCos subsystems such as the C library and POSIX compatibility layer to provide calendar time functionality.

Interrupt controller

eCos manages the on-chip Vectored Interrupt Controller (VIC). The VIC is only configured to use interrupts in non-vectored mode.

Timers

Timer 0 is used to implement the eCos system clock. If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then timer 1 is reserved for use by the profiler. Any remaining timers are available for application use.

I²C Bus

The variant HAL contains a driver for the on-chip I²C bus devices. Platform HALs need to define the clock speed and lines to be used for SDA and SCL using the following options:

`CYGPKG_HAL_ARM_LPC2XXX_I2CX`

This is the master component, enabling this activates all the other configuration options and causes the driver to create the data structures to access this bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_CLOCK`

Bus clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SDA`

This option describes the pin used for SDA on this bus. This takes the form of an invocation of the macro `__LPC2XXX_PINSEL_FUNC`. Parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2468 user manual for details of which pins may be used by each bus.

CYGPKG_HAL_ARM_LPC2XXX_I2CX_SCL

This option describes the pin used for SCL on this bus. Like SDA this takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

Note that "I2CX" is a placeholder for a given bus instance: "I2C0", "I2C1" or "I2C2".

SPI Bus

The on-chip SSP SPI devices (not the Legacy SPI device) are supported by the NXPSSP driver package, CYGPKG_DEVS_SPI_ARM_NXPSSP. This needs some configuration in the platform HAL:

CYGPKG_HAL_ARM_LPC2XXX_SPI

This is the master component, enabling this activates all the other configuration options. It also causes `ea_lpc2468_spi.c` to be compiled, which contains descriptions of the devices on the SPI buses.

CYGPKG_HAL_ARM_LPC2XXX_SPIX

This is the master component for each bus. Enabling this activates the other configuration options for this bus, and causes the driver to support this bus.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_SCLK

This option describes the pin used for SCLK on SPIX. It takes the form of an invocation of `__LPC2XXX_PINSEL_FUNC`. The parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2468 user manual for details."

CYGPKG_HAL_ARM_LPC2XXX_SPIX_MISO

This option describes the pin used for MISO on SPIX. Like SCLK it takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_MOSI

This option describes the pin used for MOSI on SPIX. Like SCLK it takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_CS_PINS

This defines the pins to be used as chip selects for this bus. It is a comma separated list of GPIO pin names, the first for device 0, the second for device 1, and so on. Pin names are defined in the `var_io.h` header in the LPC2xxx variant HAL.

Note that "SPIX" is a placeholder for a given bus instance: "SPI0" or "SPI1".

Other

Other on-chip devices (SPI, PWM, A/D converter, etc.) are not touched by the LPC2xxx variant HAL and unless used by the platform HAL are free for use for applications.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This section covers any remaining items of note related to the LPC2xxx variant support, not covered in previous sections.

LEDs

If a platform port has support for display values on LEDs, that support is standardised to be accessible from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

PLL configuration

CDL variables related to the PLL configuration are standardised across LPC2xxx implementations. The platform HAL must provide the input oscillator frequency (CYGNUM_HAL_ARM_LPC2XXX_OSC_FREQ), as well as the desired PLL multipliers and dividers selected by the user and relevant for the chosen part (CYGNUM_HAL_ARM_LPC2XXX_PLL_MULTIPLIER and CYGNUM_HAL_ARM_LPC2XXX_PLL_DIVIDER). It must also supply the VPB divider (CYGNUM_HAL_ARM_LPC2XXX_VPB_DIVIDER) which divides down the core clock (CCLK) to generate the peripheral clock (PCLK). Where applicable a platform may also define a CCLK divider (CYGNUM_HAL_ARM_LPC2XXX_CCLK_DIVIDER).

As a result, the variant HAL calculates CDL options for the absolute values of CCLK (CYGNUM_HAL_ARM_LPC2XXX_CCLK_SPEED) and PCLK (CYGNUM_HAL_ARM_LPC2XXX_PCLK_SPEED) which are exported to the rest of the system, and accessible to applications from <pkgconf/hal.h>.

It is still the responsibility of the platform HAL to initialize the PLL, although assembler helper macros are provided in <cyg/hal/var_io.h to ease implementation.

LPC2xxx definitions

The LPC2xxx variant HAL port includes the header file `var_io.h` which provides useful register definitions used by eCos, but can also be freely used by applications. It includes register definitions for subsystems unused by eCos.

It may be found in the `include/cyg/hal` directory relative to your configuration's install tree, or alternatively in the source repository at `hal/arm/lpc2xxx/var/VERSION/include/var_io.h`. However it should be properly included by applications by using:

```
#include <cyg/hal/hal_io.h>
```

to allow for platform HALs to augment or override any relevant definitions.

Power control

The kernel idle thread is scheduled to run when the system has no other tasks able to run. The idle thread can call a HAL supplied macro to place the chip into an appropriate power saving mode instead of just going around a busy loop. The LPC2xxx variant HAL defines the `HAL_IDLE_THREAD_ACTION` macro to use the LPC2xxx power control support to place the chip into `IDLE` mode which will stop the processor clock, without disabling the on-chip peripherals. This state continues until an interrupt is received. This mode has no deleterious effect on program execution, however it has been known to interfere with JTAG/ICE hardware debuggers. Therefore the CDL option `CYGIMP_HAL_ARM_LPC2XXX_IDLE_THREAD_USES_IDLE` exists in the variant HAL to ensure the processor does not enter the idle mode from the idle thread. It is recommended this option be disabled if hardware debugging solutions are used, especially if reliability is erratic.

Unless specified otherwise by the platform HAL port, no other power saving features are used and no peripherals are disabled, even those unsupported by the eCos LPC2xxx variant port. Therefore if the application wishes to conserve power it is its responsibility to place the relevant peripherals into power down modes.

Memory Acceleration Module support

The variant HAL supplies helper macros to the platform HALs to centralise initialisation code for common sub-systems, including the Memory Acceleration Module (MAM).

However it is known that there are errata which affect the MAM, and specifically restrict what mode the MAM should operate in. In some cases, such as the LPC2148, there are conflicting errata, that recommend that the MAM be used only in `Full` mode in some cases to avoid one erratum, or only `Partial` mode in others to avoid a different erratum.

Therefore the choice of MAM mode is left to the user and can be set with the CDL option `CYGHWR_HAL_ARM_LPC2XXX_MAM_MODE` in the variant HAL.

Virtual Vector support

As described in the common HAL documentation, virtual vectors are used to abstract certain services which could be shared between a resident ROM monitor, and an application. In the case where an application is entirely stand-alone, and does not use any resident ROM monitor - for example, if it is itself a ROM application - then virtual vector support is unnecessary.

The CDL configuration option `CYGFUN_HAL_LPC2XXX_VIRTUAL_VECTOR_SUPPORT` can be used to control the presence of virtual vectors, although it is expected that the default value will be selected appropriately in any case. Disabling virtual vector support can save both Flash and RAM use, which can be important for targets with restricted memory.

X. Keil MCB2387 Board Support

Overview

Name

eCos Support for the Keil MCB2387 Board — Overview

Description

The Keil MCB2387 Board is fitted with an NXP LPC2387 processor rated up to 72MHz, which contains 64KB of SRAM and 512KB of FLASH. It provides access to two on-chip UARTs, an MMC/SD card socket, and a PHY connected to the on-chip Ethernet MAC. Refer to the board documentation for full details.

For typical eCos development, a GDB StubROM image is programmed into the LPC2387 on-chip flash memory, and the board will boot this image from reset. This provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

This documentation describes platform-specific elements of the MCB2387 Board support within eCos. Documentation on the [NXP LPC2xxx variants](#) is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The MCB2387 board has 512Kbyte of on-chip Flash memory. In a typical setup, the stubrom will run from this internal flash. An image must be programmed into this flash using either the FlashMagic utility, or via a JTAG debugger.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2387 memory mapping control to location 0x00000000 for speed of interrupt vector processing. The rest of SRAM is available for use by the application.

The NXP LPC2xxx variant HAL includes support for the on-chip serial devices which is [documented in the variant HAL](#). While the interrupt-driven serial driver supports the line status and modem control features of the UART devices, none of these lines are made available on the COM0 or COM1 connectors.

The MCB2387 board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the [LPC2xxx variant HAL](#).

The on-chip Ethernet MAC is supported.

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the OEM board.

Drivers for I2C and SPI are present. However, since there are no on-board devices connected to these busses, they have only been tested using external devices attached to the board for the purpose.

Tools

The MCB2387 board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Setup

Name

Setup — Preparing the MCB2387 Board for eCos Development

Overview

In a typical development environment, the MCB2387 board boots from internal flash into the GDB stubrom. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable image into flash memory.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Embedded Systems Academy Flash Magic utility is available. Install Flash Magic from <http://www.flashmagictool.com>. Connect a 9-pin serial cable from a COM port on the PC to the COM0 connector on the MCB2387 Board. Power the board via a USB cable.

Set the ISP jumpers (J9(RST) and J10(ISP) on, J13(ETM) off) and press the reset button. The board is now running a special NXP boot loader. Start Flash Magic and set the Communications section to select the COM port used above, 38400 baud, device LPC2387, Interface “None (ISP)” and 12MHz Oscillator Frequency. Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the serial cable is connected, the ISP jumpers are installed and the correct COM port is being used.

Check “Erase blocks used by Hex File” under “Erase”. In the “Hex File” section, select the `stubrom.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the upload.

When the process completes, the utility should be closed. Reset the ISP jumpers (J9(RST) and J10(ISP) off, J13(ETM) on). Verify the programming has been successful by starting a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. The output should be similar to the following:

```
+ $T050f:6c080000;0d:b8070040;#53
```

This is the stubrom reporting it's state using the GDB remote protocol.

Rebuilding the Stubrom

Should it prove necessary to rebuild the Stubrom binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild RedBoot for the MCB2387 are:

Setup

```
$ mkdir stub_mcb2387_rom
$ cd stub_mcb2387_rom
$ ecosconfig new mcb2387 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.hex`.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The mcb2387 platform HAL package is loaded automatically when eCos is configured for an mcb2387 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The mcb2387 platform HAL package supports three separate startup types:

JTAG

This is the startup type which is normally used during JTAG based application development. arm-elf-gdb is then used to connect to the JTAG device and load a JTAG startup application into memory and debug it. It is assumed that the basic hardware has already been initialized via the JTAG device's initialization script. Otherwise the application is entirely self contained and should contain drivers for all hardware used.

RAM

This is the startup type which is normally used during stubrom based application development. The board has the stubrom programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the basic hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubrom, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the stubrom.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

Flash Driver

The `CYGPKG_DEVS_FLASH_LPC2XXX` package contains all the code necessary to support the on-chip flash.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 72MHz. However, the platform HAL currently sets the clock to 48MHz, duplicating the configuration in the supplied example code as a consequence of CPU errata affecting various revisions of the LPC2387. Setting the CPU revision with the `CYGHWR_HAL_ARM_LPC2XXX_MCB2387_CPU_REVISION` configuration option can be used to provide default clock settings appropriate to the CPU revision in use. If the CPU revision cannot be guaranteed it should be left as "Initial". The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

I²C Bus Configuration

The on-chip I²C devices are supported by a driver in the variant HAL package. Each bus for this driver needs to be configured in the platform HAL with the following options:

`CYGPKG_HAL_ARM_LPC2XXX_I2CX`

This is the master component, enabling this activates all the other configuration options and causes the driver to create the data structures to access this bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_CLOCK`

Bus clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SDA`

This option describes the pin used for SDA on this bus. This takes the form of an invocation of the macro `__LPC2XXX_PINSEL_FUNC`. Parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2387 user manual for details of which pins may be used by each bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SCL`

This option describes the pin used for SCL on this bus. Like SDA this takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

Note that "I2CX" is a placeholder for a given bus instance: "I2C0", "I2C1" or "I2C2". By default the platform HAL does not enable any I²C buses since there are no on-board devices.

SPI Bus Configuration

The on-chip SSP SPI devices (not the Legacy SPI device) are supported by the NXPSSP driver package, `CYGPKG_DEVS_SPI_ARM_NXPSSP`. This needs some configuration in the platform HAL:

`CYGPKG_HAL_ARM_LPC2XXX_SPI`

This is the master component, enabling this activates all the other configuration options. It also causes `mcb2387_spi.c` to be compiled, which contains descriptions of the devices on the SPI buses.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX`

This is the master component for each bus. Enabling this activates the other configuration options for this bus, and causes the driver to support this bus.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_SCLK`

This option describes the pin used for SCLK on SPIX. It takes the form of an invocation of `__LPC2XXX_PINSEL_FUNC`. The parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2387 user manual for details."

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_MISO`

This option describes the pin used for MISO on SPIX. Like SCLK it takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_MOSI`

This option describes the pin used for MOSI on SPIX. Like SCLK it takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

`CYGPKG_HAL_ARM_LPC2XXX_SPIX_CS_PINS`

This defines the pins to be used as chip selects for this bus. It is a comma separated list of GPIO pin names, the first for device 0, the second for device 1, and so on. Pin names are defined in the `var_io.h` header in the LPC2xxx variant HAL.

Note that "SPIX" is a placeholder for a given bus instance: "SPI0" or "SPI1". By default the platform HAL does not enable any SPI busses since there are no on-chip devices.

MCI peripheral configuration

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the board. This support is provided in conjunction with the generic MMC/SD driver package (`CYGPKG_DEVS_DISK_MMC`), the Primecell MCI driver package (`CYGPKG_DEVS_MMCSL_ARM_PRIMECELL_MCI`) and the LPC2xxx variant HAL in order to provide some elements of the DMA support. Documentation and configuration options within those packages should also be consulted.

The following CDL configuration options are used to control the behaviour of the MMC/SD card support:

MMC/SD card support (`CYGPKG_HAL_ARM_LPC2XXX_MCB2387_MCI`)

This option allows the MMC/SD card support as a whole to be enabled or disabled, although the generic disk device driver package (`CYGPKG_IO_DISK`) must be loaded in order to enable the MMC/SD support.

Use on-chip USB memory for DMA (CYGSEM_HAL_ARM_LPC2XXX_MCB2387_MCI_USE_USB_MEM_FOR_DMA)

The LPC2387 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2387 have errata which decreases their achievable CPU clock frequency.

Using on-chip memory dedicated to USB helps reduce or remove these problems, depending on CPU frequency. Clearly this option must be disabled if the on-chip USB peripheral is to be used. It is also desirable to disable this option if the CPU frequency is high enough, in order to remove an extra copy on every data transfer, thus improving performance. The USB memory used is 512 bytes at the start of the USB memory space (0x7FD00000).

If this option is disabled and the DMA is not able to proceed quickly enough, this will be visible in the form of I/O errors. In that case, if it is not possible to enable this option it is recommended to adjust the CYGDAT_HAL_ARM_LPC2XXX_MCB2387_MCI_BUS_SPEED_LIMIT configuration option.

Lock AHB bus during DMA transfer (CYGSEM_HAL_ARM_LPC2XXX_MCB2387_MCI_DMA_LOCKS_AHB)

The AMBA Hardware Bus (AHB) is used to connect AMBA peripherals within the LPC2387, including the ARM core, DMA controller and memory controllers. When this option is enabled, the AHB is locked for the duration of MCI DMA transfer bursts. If another AMBA host needs to make a transfer it may be delayed as a result, which may not be desirable.

Disabling this option allows the AHB arbiter to permit other AHB hosts to perform transfers. Of course this may mean the MCI DMA transfers can in turn themselves get delayed, risking data overruns or underruns in MCI transfers, resulting in I/O errors during block reads or writes. This is particularly likely on processors running at slower clock speeds where there may already be difficulties with the DMA servicing data transfers quickly enough.

MMC/SD bus frequency limit (CYGNUM_HAL_ARM_LPC2XXX_MCB2387_MCI_BUS_SPEED_LIMIT)

The LPC2387 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2387 have errata which decreases their achievable CPU clock frequency. The adjacent options to use on-chip USB memory and to lock the AHB bus can help prevent this, but sometimes they are insufficient to prevent data overruns or underruns resulting in I/O errors during block reads or writes. In which case the only remaining recourse is to reduce the required data transfer rate between the MCI and the card.

This option can be used to impose an upper limit on the MMC/SD bus frequency. The value used in this option is measured in Hertz, and the use of 4-bit mode with SD cards is not a factor - this option provides the bus frequency, so a 4-bit bus will transfer four times the amount of data as a 1-bit bus in the same time period.

Note that this option provides a limit, and does not mean the card bus will operate at that frequency. The frequency is also governed by what the card will support, and the resolution of the clock used to derive the MMC/SD clock signal, and how it can be divided down.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MCB2387 board hardware, and should be read in conjunction with that specification. The LPC2387 platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor to do most of this and for JTAG startup, where some initialization will be done by the JTAG device.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks); Memory Mapping control registers to map SRAM to 0x0 and the Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x80000. The MAM is enabled to accelerate memory reads from this area. A driver is available for using this flash via the eCos flash API.

internal SRAM

This is located at address 0x40000000 of the memory space, ending at location 0x4000FFFF. The first 64 bytes are mapped to location 0x00000000.

on-chip peripherals

These are accessible via location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2387 User Manual.

XI. Phyttec phyCORE LPC2294 Board Support

Overview

Name

eCos Support for the Phytex phyCORE LPC2294 Board — Overview

Description

The Phytex phyCORE LPC2294 Board is fitted with a Philips LPC2294 processor rated to 60MHz, which contains up to 64KB of SRAM and up to 256KB of FLASH. When used in conjunction with the phyCORE HD200 development board, it provides two 9-pin RS-232 serial interfaces connected to the LPC2294 on-chip UARTs, a single LED, two CAN interfaces and an SMSC LAN91C111 ethernet interface. Refer to the board documentation for full details.

For typical eCos development, a RedBoot image is programmed into the LPC2294 on-chip flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0 or via the ethernet.

This documentation describes platform-specific elements of the phyCORE LPC2294 Board support within eCos. Documentation on the [Philips LPC2xxx variants](#) is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The phyCORE LPC2294 Board has 128Kbyte of on-chip Flash memory. In a typical setup, RedBoot will load and run from this internal flash. An initial image must be programmed into this flash using either the FlashMagic utility, or via a JTAG debugger. Following this, it may be reprogrammed using flash drivers in RedBoot.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2294 memory mapping control to location 0x00000000 for speed of interrupt vector processing. The rest of SRAM is available for use by the application. One MByte of SRAM is available at 0x81000000; the first 64KBytes of this is reserved for use by RedBoot, the rest is available for the code and data of loaded applications.

The Philips LPC2xxx variant HAL includes support for the two on-chip serial devices and is [documented in the variant HAL](#). The interrupt-driven serial driver supports the line status and modem control (including hardware handshaking) lines on UART1 only. These handshaking lines are not accessible at the DB9 connector (P1B).

The phyCORE LPC2294 Board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the [LPC2xxx variant HAL](#).

The SMSC LAN91C111 ethernet MAC is supported. However, due to PCB tracking problems, it is only capable of running at 10MBit/s and the driver forces the device to only operate at that speed.

Tools

The phyCORE LPC2294 Board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils

Overview

version 2.14.

Setup

Name

Setup — Preparing the phyCORE LPC2294 Board for eCos Development

Overview

In a typical development environment, the phyCORE LPC2294 Board boots from internal flash into RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable stubrom image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.hex
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.srec

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the stubrom.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Embedded Systems Academy Flash Magic utility is available.

Install Flash Magic from <http://www.flashmagictool.com>.

Connect the RS232 cable supplied with the phyCORE LPC2294 between port 0 of the phyCORE LPC2294 and the host PC. Apply power to the board and with the Boot button (S_1) held down, press and release the reset button (S_2). The board is now running a special NXP boot loader. Start Flash Magic and set the Communications section to select the appropriate serial port, 38400 baud, device LPC2294, Interface “None (ISP)” and 12MHz Oscillator Frequency. Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the serial cable is connected correctly, that the board was booted with the Boot button (S_1) held down and that the correct communication parameters have been selected in Flash Magic.

Check “Erase blocks used by Hex File” under “Erase”. In the “Hex File” section, select the `redboot_ROM.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the upload.

When the process completes, the utility should be closed. Verify that programming has been successful by starting a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and RedBoot should start. The output should be similar to the following:

```
+... Read from 0x801e0000-0x801fffff to 0x810e0000:
... Read from 0x801ff000-0x801fffff to 0x810df000:
... waiting for BOOTP information
Ethernet eth0: MAC address 00:50:c2:32:ad:40
IP: 10.0.0.200/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.102, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 17:10:02, Dec  1 2004

Platform: Phytex phyCORE LPC229x Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x81000000-0x81100000, [0x8100b1a8-0x810dd000] available
FLASH: 0x80000000 - 0x801fffff 16 x 0x20000 blocks
RedBoot>
```

If it is ever necessary to reinstall RedBoot, the above directions can be repeated. Alternatively, a new RedBoot may be installed from RedBoot itself. It is not possible to do this directly, since RedBoot is executing from the flash that needs to be erased and reprogrammed. Instead it is necessary to run a RAM version of RedBoot, use that to download the new ROM RedBoot to RAM, and then program that to flash.

The following shows an example session to do this. It assumes that `redboot_RAM.srec` and `redboot_ROM.bin` are available via TFTP on the server set up in **fconfig**.

```
RedBoot> load redboot_RAM.srec
Using default protocol (TFTP)
Entry point: 0x81010040, address range: 0x81010000-0x8102c04c
RedBoot> go
+Ethernet eth0: MAC address 00:50:c2:3b:aa:9d
IP: 192.168.7.251/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.5
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [RAM]
eCosCentric certified release, version v2_0_105 - built 15:42:30, Dec  5 2008

Platform: Phytex phyCORE LPC229x Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0x81000000-0x81800000, [0x81035f90-0x817dd000] available
FLASH: 0x00000000-0x0003dfff, 8 x 0x2000 blocks, 2 x 0x10000 blocks, 7 x 0x2000s
FLASH: 0x80000000-0x80ffffff, 63 x 0x20000 blocks, 8 x 0x4000 blocks, 63 x 0x20s
RedBoot> load -r -b ${freememlo} redboot_ROM.bin
Raw file loaded 0x81036000-0x81053463, assumed entry at 0x81036000
RedBoot> fis write -f 0x00000000 -b ${freememlo} -l 0x20000
```

```

* CAUTION * about to program FLASH
      at 0x00000000..0x0001ffff from 0x81036000 - continue (y/n)? y
... Erase from 0x00000000-0x0001ffff: .....
... Program from 0x81036000-0x81056000 to 0x00000000: .....
RedBoot> reset
+Ethernet eth0: MAC address 00:50:c2:3b:aa:9d
IP: 192.168.7.251/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.5
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v2_0_105 - built 15:42:52, Dec  5 2008

Platform: Phytect phyCORE LPC229x Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0x81000000-0x81800000, [0x8100acc8-0x817dd000] available
FLASH: 0x00000000-0x0003dfff, 8 x 0x2000 blocks, 2 x 0x10000 blocks, 7 x 0x2000s
FLASH: 0x80000000-0x80ffffff, 63 x 0x20000 blocks, 8 x 0x4000 blocks, 63 x 0x20s
RedBoot>

```

Rebuilding RedBoot

Should it prove necessary to rebuild the RedBoot binary, this is done most conveniently at the command line. Assuming your PATH and ECOS_REPOSITORY environment variables have been set correctly, the steps needed to rebuild RedBoot are:

```

$ mkdir redboot_phycore_rom
$ cd redboot_phycore_rom
$ ecosconfig new phycore_lpc2294 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/lpc2xxx/phycore_lpc229x/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make

```

At the end of the build the install/bin subdirectory should contain the file `redboot.hex`.

Note: The PhyCORE LPC2294 board can be fitted with a wide range of flash and SRAM parts. So it may be necessary to adjust the configuration after importing the `redboot_ROM.ecm` file to match the hardware being used. The [Memory Configuration](#) section contains full details of the options available for this.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The phycore platform HAL package is loaded automatically when eCos is configured for an `phycore_lpc2294` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The phyCORE LPC229x platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubrom. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubrom, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

Flash Driver

The phyCORE LPC2294 board contains a number of AMD flash devices. The CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2 package contains all the code necessary to support these parts and the CYGPKG_DEVS_FLASH_ARM_PHYCORE package contains definitions that customize the driver to the phyCORE LPC2294 board.

Ethernet Driver

The phyCORE-LPC229x board contains an SMSC LAN91C111 ethernet MAC. The CYGPKG_DEVS_ETH_SMSC_LAN91CXX package contains all the code necessary to support this device and the CYGPKG_DEVS_ETH_ARM_PHYCORE package contains definitions that customize the driver to the phyCORE LPC2294 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_DENOMINATOR which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

Memory Configuration

The PhyCORE LPC2294 board can be fitted with a wide range of flash and SRAM parts. The following options adjust the configuration of eCos and RedBoot to accommodate these variations:

CYGHWR_HAL_ARM_LPC2XXX_PHYCORE_MEMORY_CONFIGURATION_FLASH

This option describes the flash devices fitted to the board. Possible values are: AM29DL800BT, AM29LV800BT, AM29LV160BT and AM29LV320BT. Of these only the AM29DL800BT and AM29LV320BT variants have been tested.

CYGHWR_HAL_ARM_LPC2XXX_PHYCORE_MEMORY_CONFIGURATION_FLASH_COUNT

This option defines the number of flash devices fitted. Flash devices can only be fitted in pairs, and there is only space for up to 4, so this value can only be 2 or 4.

CYGHWR_HAL_ARM_LPC2XXX_PHYCORE_MEMORY_CONFIGURATION_SRAM_SIZE

This option defines the total SRAM size. The board can be fitted with two or four SRAM devices, of 512KB, 1MB or 2MB each, giving Possible possible SRAM sizes of: 0x100000, 0x200000, 0x400000 or 0x800000.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the phyCORE LPC2294 Board hardware, and should be read in conjunction with that specification. The phyCORE LPC229x platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks); Memory Mapping control registers to map SRAM to 0x0; the memory controller for access to external FLASH, SRAM and ethernet; and the Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x40000. The MAM is enabled to accelerate memory reads from this area. A driver is available for using this flash via the eCos flash API.

external Flash

This is located at address 0x80000000 of the memory space. It is not used by default by eCos, although if RedBoot is asked to manage the Flash, it reserves flash addresses 0x801E0000 thru 0x801FF000. If RedBoot stores its configuration data in Flash, then addresses 0x801FF000 thru 0x801FFFFFF are reserved by RedBoot. RedBoot also reserves the first block of Flash at 0x80000000 thru 0x80001FFFF to ensure that it remains erased and does not therefore inhibit the execution of RedBoot from the internal Flash. External flash is 32bits wide and accessed with 7 wait states.

internal SRAM

This is located at address 0x40000000 of the memory space, ending at location 0x40004000. The first 64 bytes are mapped to location 0x00000000.

external SRAM

This is located at address 0x81000000 of the memory space, ending at location 0x81100000. For RAM startup, available SRAM starts at location 0x81010000, with the bottom 64Kbytes reserved for use by RedBoot.

on-chip peripherals

These are accessible via location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2294 User Manual.

Other Issues

The LEDs may be accessed from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Values from 0 to 16 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off, and with P0.7 being the MSB, and P0.4 the LSB.

The LEDs are also used during platform initialization and only P0.4 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

XII. Ashling EVBA7 Eval Board Support

Overview

Name

eCos Support for the Ashling EVBA7 Eval Board — Overview

Description

The Ashling EVBA7 Eval Board is fitted with a Philips LPC2000 processor rated to 60MHz, which contains up to 64KB of SRAM and up to 256KB of FLASH. The board has two 9-pin RS-232 serial interfaces connected to the LPC2000 on-chip UARTs, an LED bank and JTAG/USB debug interfaces. Refer to the board documentation for full details.

The standard EVBA7 is fitted with an LPC2106 microcontroller. Some versions of the EVBA7 board are fitted with an adaptor that either contains a specific LPC2000 part, or a socket into which one of several LPC2000 parts may be fitted.

For typical eCos development, a RedBoot or GDB Stubrom image is programmed into the LPC2000 on-chip flash memory, and the board will boot this image from reset. Both RedBoot and the GDB stub ROM provide GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

This documentation describes platform-specific elements of the EVBA7 Eval Board support within eCos. Documentation on the [Philips LPC2xxx variants](#) is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The EVBA7 Eval Board has up to 128Kbyte of on-chip Flash memory. In a typical setup, RedBoot or the GDB Stubrom will load and run from this internal flash. No support for managing internal Flash is included in this port - the Ashling FlashLPC Utility is required to program the internal Flash. 24Kbytes of internal flash memory should be reserved for the GDB Stubrom, the remainder being free for the application's use. RedBoot will occupy the entire internal ROM.

EVBA7 boards fitted with the PA-EVBA7-144 adaptor also have 1MByte of external RAM on the adaptor. In this case, eCos is configured to use this memory rather than the internal SRAM. These are also the only boards capable of running RedBoot.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2000 memory mapping control to location 0x00000000 for speed of interrupt vector processing. SRAM from location 0x40000040 to 0x40001000 is used by the GDB Stubrom. The rest of SRAM is available for use by the application.

The Philips LPC2xxx variant HAL includes support for the two LPC2000 on-chip serial devices and is [documented in the variant HAL](#). The interrupt-driven serial driver supports the line status and modem control (including hardware handshaking) lines on UART1 only.

The EVBA7 Eval Board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the [LPC2xxx variant HAL](#).

Tools

The EVBA7 Eval Board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14.

Setup

Name

Setup — Preparing the EVBA7 Eval Board for eCos Development

Overview

In a typical development environment, the EVBA7 Eval Board boots from internal flash into the GDB stubrom monitor or RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable ROM or RedBoot image into flash memory.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the stubrom or RedBoot.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Ashling FlashLPC Utility installed is available. The first step is to connect the RS232 cable supplied with the EVBA7 between port 0 of the EVBA7 and the host PC. Now install the jumper labelled “ENABLE SERIAL ISP”. On the base EVBA7 this is jumper JP6; on boards fitted with one of the FA-EVBA7 adaptors, this is JP5 on the adaptor; on boards fitted with a PA-EVBA7 adaptor, this is JP3 on the adaptor. Refer to the board documentation for full details. Apply the power, or press the reset button.

The board is now running a special Philips boot loader. Start the FlashLPC Utility, and ensure that the selected device matches the device installed on the EVBA7. Choose the appropriate COM port that is being used on your PC and select 115200 baud for both the initial and final baud rates. Set the Crystal KHz value to 14745, ensure that the stop bits value is set to 1 and the packet size is set to 100%. Establish communication with the board by pressing the “Connect” button.

Now in the “Flash Programming” section, select the `stubrom.srec` or `redboot.srec` file. Set the file format to “S-Record Format”, select “Erase individual sectors before programming” under the Programming Options and ensure that “Automatically add checksum to vector table” is ticked. Finally press “Program”, or “Program and Verify” to program the ROM image.

When the process completes, remove the “ENABLE SERIAL ISP” jumper. Verify the programming has been successful by starting a terminal emulation application such as HyperTerminal on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. For boards programmed with GDB stubs the output should be similar to the following:

```
+ $T050f:ec070000;0d:28080040;#52
```

This is the stubrom attempting to communicate with GDB and indicates that it is functioning correctly.

For boards fitted with RedBoot, you should see the RedBoot startup messages ending with a RedBoot prompt.

Rebuilding GDB Stubrom

Should it prove necessary to rebuild the Stubrom binary, this is done most conveniently at the command line. Assuming your PATH and ECOS_REPOSITORY environment variables have been set correctly, the steps needed to rebuild the stubrom are:

```
$ mkdir stub_rom
$ cd stub_rom
$ ecosconfig new evba7 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `stubrom.srec`.

Rebuilding RedBoot

Should it prove necessary to rebuild the RedBoot binary, this is done most conveniently at the command line. Assuming your PATH and ECOS_REPOSITORY environment variables have been set correctly, the steps needed to rebuild RedBoot are:

```
$ mkdir evba7_redboot
$ cd evba7_redboot
$ ecosconfig new evba7_2294 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/lpc2000/evba7/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.srec`.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The evba7 platform HAL package is loaded automatically when eCos is configured for a evba7 target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The evba7 platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has the GDB stubrom or RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. arm-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the ROM monitor, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

The ROM Monitor and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubrom or RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the EVBA7 Eval Board hardware, and should be read in conjunction with that specification. The EVBA7 Eval Board platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks), Memory Mapping control registers to map SRAM to 0x0, and Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x20000. No flash driver is provided for the on-chip Flash. The MAM is enabled to accelerate memory reads from this area.

internal SRAM

This is located at address 0x40000000 of the memory space, and is 16, 32 or 64k in size, depending on the chip fitted. The first 64 bytes are mapped to location 0x0000000. When this is the only RAM available, the virtual vector table starts at 0x40000050 and extends to 0x40000150. The remainder of SRAM is available for use by ROM based applications. For RAM startup applications, SRAM below 0x40001000 is reserved for the GDB stubrom and the remainder is available for the application.

On boards fitted with the PA-EVBA7-144 adaptor and where the external SRAM is being used, only the first 64 bytes are used as described above. The remainder of internal SRAM is not used by eCos.

external SRAM

This SRAM is only present if the EVBA7 is fitted with a PA-EVBA7-144 adaptor. It is located at address 0x81000000 of the memory space, and is 1MByte in size. When this memory is being used for applications the virtual vector table starts at 0x81000050 and extends to 0x81000150. The remainder is available for use by ROM based applications. For RAM startup applications, memory below 0x81010000 is reserved for RedBoot and the remainder is available for the application.

on-chip peripherals

These are accessible at location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2000 User Manual.

Other Issues

The LEDs may be accessed from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Values from 0 to 16 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off, and with P0.7 being the MSB, and P0.4 the LSB.

The LEDs are also used during platform initialization and only P0.4 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

XIII. Embedded Artists QuickStart Board Support

Overview

Name

eCos Support for the Embedded Artists QuickStart Boards — Overview

Introduction

This platform HAL is designed to support the QuickStart board series from Embedded Artists, fitted with an NXP LPC2xxx microcontroller, and optionally connected to an Embedded Artists QuickStart Prototype Board.

The support for the QuickStart board series provided by this HAL has been initially developed for the Embedded Artists (henceforth 'EA') LPC2148 USB QuickStart Board. This HAL documentation therefore presently corresponds to that particular board instance, and future supported variants will cause this documentation to be updated accordingly.

Description

The Embedded Artists QuickStart Board is fitted with an NXP LPC2xxx processor rated at up to 60MHz, which contains up to 64KB of SRAM and up to 512KB of FLASH, depending on choice of LPC2xxx variant. The board itself has a single 9-pin RS-232 serial interface connected to the LPC2xxx on-chip UART 0, an I2C EEPROM, and a USB device interface. Refer to EA's QuickStart board documentation for full details.

Further peripheral support is available if the board is mounted on to the EA QuickStart Prototype Board, including push buttons, LEDs, JTAG, MMC/SD card socket, buzzer, 7-segment LED, and a further 9-pin RS-232 serial interface connected to the LPC2xxx on-chip UART 1.

The typical mode of operation for eCos development usually depends on the amount of memory available. On LPC2xxx variants with 64Kbytes or more of SRAM, a GDB stub ROM image is programmed into the LPC2xxx on-chip flash memory, and the board will boot this image from reset. While RedBoot may also be used, its larger RAM footprint requirements usually make it unsuitable. Both RedBoot and the GDB stub ROM provide GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

On LPC2xxx variants with less than 64Kbytes SRAM, such as the 32Kbytes on the LPC2148, it is typically expected that standalone applications will be programmed directly to on-chip Flash, either using a hardware JTAG/ICE unit via the QuickStart Prototype Board, or by serial using the on-chip In-System Programming (ISP) mechanism included with NXP LPC2xxx microcontrollers and a suitable host application running on a PC.

This documentation describes platform-specific elements of the EA QuickStart Board support within eCos. Documentation on the [NXP LPC2xxx variants](#) is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The NXP LPC2xxx microcontrollers on the EA QuickStart Boards have up to 512Kbytes of on-chip Flash memory. In a typical setup, the GDB stub ROM or the user application will load and run from this internal flash. For initial

programming of the internal Flash, external support is required, such as the NXP LPC2000 Flash Utility, the Flash Magic tool, or a hardware JTAG/ICE unit. The latter may be used with its own in-built LPC2xxx flash programming support if it exists, or the eCosPro® **ecoflash** utility. 28Kbytes of internal flash memory should be reserved for the GDB Stub ROM, the remainder being free for the application's use. Note that the LPC2xxx primary boot loader and IAP code reside in boot blocks located at the end of on-chip Flash. To determine the number and size of blocks reserved for their use, consult the specific LPC2xxx variant's datasheet.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2xxx memory mapping control to location 0x00000000. When loading applications using the GDB stub ROM, SRAM from location 0x40000040 to 0x40001000 is reserved for its use. The rest of SRAM is generally available for use by the application. Programs booted from ROM, or loaded directly into SRAM via JTAG may use all SRAM. In all cases, if using the eCos LPC2xxx Flash driver, the last 32 bytes (or more if a separate program buffer is used) become reserved due to the requirements of the IAP code.

The NXP LPC2xxx variant HAL includes support for the two LPC2xxx on-chip serial devices and is [documented in the variant HAL](#). Although the interrupt-driven serial driver supports the line status and modem control lines on UART 1 (UART 0 not having such support), the QuickStart boards do not connect these pins, and so that functionality is unavailable.

The EA QuickStart port includes support for the on-chip watchdog, RTC (wallclock), interrupt controller (VIC) and on-chip Flash. This support is documented in the [LPC2xxx variant HAL](#).

Tools

The QuickStart Board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was created using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Setup

Name

Setup — Preparing the EA QuickStart Board for eCos Development

Overview

In a typical development environment, the EA QuickStart Board boots from internal flash into either the GDB stub ROM monitor or directly into the user application. In the case of microcontrollers with less than 64Kbytes of SRAM, the latter is recommended. eCos applications to be loaded and run from the GDB stub ROM monitor may be configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable ROM image into Flash memory, either the GDB stub ROM or application images.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the application, or if applicable, GDB stub ROM. A "straight through" 9-pin RS232 serial cable, with Male<->Female connectors is required. Using a "null modem" serial cable will not work.

Initial Installation

Board setup

Jumper settings must be checked and potentially changed on the board to ensure correct operation. This section describes jumper settings that are known to require attention. In general, any board-specific documentation from Embedded Artists takes precedence over the documentation here, as this may reflect hardware which has been modified since the time of writing of this documentation. Most QuickStart boards are very similar to each other, the only change of note being of course the choice of LPC2xxx microcontroller fitted. But if your board does not fit the description here (which has initially been based on the LPC2148 USB QuickStart) then you should consult the board documentation.

Firstly, there are two jumpers located close to the serial connector. In general, these jumpers should only be closed (i.e. jumper fitted and connecting the two pins) when wishing to reprogram the on-chip Flash via ISP. Otherwise they should remain open (jumper not connecting the two pins) so that any unplugging of the serial connector, movement of the serial connector, or use of flow control signals from the host PC, do not cause a spurious reset or interrupt (on the EINT1 line) of the board.

If the board is being powered directly by USB, then a jumper next to the USB connector should be closed. Otherwise it *must* be open. This also means the jumper must be open if the QuickStart board is mounted on the Prototype Board, and power is being provided by either the DC power connector or USB connector on the Prototype Board.

Note that if the Prototype Board is fitted, but the QuickStart board is being powered by its USB connector as opposed to the Prototype board's USB connector (thus meaning that the above jumper would be closed), then pin P0.23 is used as a USB power indication on LPC214x boards. This prevents its use as an SPI chip select line for the 7-segment LED on the Prototype Board.

If mounting the QuickStart LPC2xxx board on the QuickStart Prototype Board, then consult the EA documentation for the correct jumper settings and socket location appropriate to the fitted LPC2xxx model. This includes settings for the JTAG connector. In the case of the LPC213x/LPC214x, the jumper labelled "JTAG" must be closed, and the jumper labelled "DBGSEL" must be open.

It may also be useful to be aware that although eCos configures the PWM pin for the buzzer, it does not directly support it, and so it is probably useful to open the jumper labelled "P0.7" to disable the buzzer.

The jumpers adjacent to the LEDs may remain in their default state of closed, in order to get insight into system operation, and to allow use of the user-configurable LEDs, as described [later](#).

Flash Installation

This process assumes that a Microsoft Windows machine with the Flash Magic utility installed is available. Flash Magic is a tool for programming flash based microcontrollers from NXP using a protocol via the RS232 serial port to communicate with the In-System Programming (ISP) firmware on the LPC2xxx. The Flash Magic utility is sponsored by NXP and available from this website (<http://www.flashmagictool.com/>).

The first step is to connect the RS232 cable between the serial port of the QuickStart board and the host PC. Do not use the serial port on the Prototype board. Now close the two jumpers adjacent to the serial port on the QuickStart board. These allow the software on the PC to reset the LPC2xxx and enter the ISP firmware. Finally apply the power.

Start the Flash Magic utility on the host PC, and a window will be displayed allowing various parameters to be configured in a series of steps. For step 1, firstly choose the appropriate COM port that is being used on your PC and set the Baud Rate to 38400 baud. Next select the appropriate LPC2xxx device in use such as LPC2148. The "Interface" should be set to "None (ISP)". And finally for step 1 choose the appropriate Oscillator Frequency for the QuickStart board in use. This may be found in the board documentation, and is usually visibly readable on the surface of the oscillator on the board (in a metal package). For example for the LPC2148 USB QuickStart, the oscillator reads 12.000 indicating 12MHz.

For step 2, it is usually adequate to leave the option "Erase blocks used by hex file" checked, and ignore the other settings. For step 3, you must select the program image to be downloaded, in Intel HEX format. To program the pre-built GDB stub ROM image, locate the file `gdb_module.hex` in the `loaders` subdirectory of your release. To generate an Intel HEX format version of an application you have built yourself run the following command at a shell prompt:

```
$ arm-elf-objcopy -O ihex app.elf app.hex
```

This converts the application image in ELF format (as output by the linker), to Intel HEX format in the file `app.hex`. Note that the arm-elf tools must be on your path at this point. If they are not, run the command:

```
$ . /opt/ecos/ecosenv.sh
```

before you run the above **arm-elf-objcopy** command.

In step 4, it is recommended to set the option "Verify after programming". Finally it is possible to click on "Start" to program the image into the on-chip Flash.

Tip: If there is a problem communicating with the board, such as a report of a failure to autobaud, then this may imply that the Flash Magic tool was not able to control the serial lines properly. This can happen with some USB-Serial converters. For Prototype Board users, to workaround this issue, power the board off and remove

(i.e. open) the two jumpers next to the serial port. Then simultaneously press the buttons marked 'Reset' and 'P0.14' on the prototype board (the latter corresponds to interrupt EINT1), then release the Reset button, and finally release the 'P0.14' button. This is an alternative mechanism of forcing the ISP firmware to be entered. Once this has been successfully performed, the programming operation may be retried from the Flash Magic utility.

When the process completes, remove (i.e. open) the two jumpers next to the serial port. If a GDB stub ROM image has been programmed, verify the programming has been successful by starting a terminal emulation application such as HyperTerminal on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. For boards programmed with GDB stubs the output should be similar to the following:

```
+ $T050f:ec070000;0d:28080040;#52
```

This is the stubrom attempting to communicate with GDB and indicates that it is functioning correctly.

Rebuilding the GDB Stub ROM

Should it prove necessary to rebuild the GDB Stub ROM binary, this is done most conveniently at the command line. Your PATH and ECOS_REPOSITORY environment variables must first be set correctly. The following steps given an example of how to rebuild the stubs for a QuickStart board with LPC2148:

```
$ mkdir stub_rom
$ cd stub_rom
$ ecosconfig new ea_quickstart_lpc2148 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the install/bin subdirectory should contain the files `gdb_module.img` (ELF format), `gdb_module.srec` (Motorola S-Record format), `gdb_module.bin` (raw binary format), and `gdb_module.hex` (Intel HEX format).

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The EA QuickStart platform HAL package is loaded automatically when eCos is configured for a target which uses it. The target names include the LPC2xxx model in use. At this time the only target supported is the `ea_quickstart_lpc2148` target. It should never be necessary to load this platform HAL package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The EA QuickStart platform HAL package supports three separate startup types:

ROM

This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. On targets with less than 64Kbytes of SRAM, this is the startup type normally used.

RAM

This is the startup type which is normally used during application development on targets with 64Kbytes of SRAM or greater. The board has the GDB stubrom or RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the ROM monitor, including diagnostic output. Larger applications may not fit into the available SRAM, in which case ROM startup may be required.

JTAG

This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading. Some sample configuration and initialisation scripts for a number of JTAG debugging solutions may be found in the `misc` subdirectory of the platform HAL package within the component repository.

The ROM Monitor and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubrom or RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup. Virtual vector support is usually only required for RAM startup applications or ROM monitors. The CDL option `CYGFUN_HAL_LPC2XXX_VIRTUAL_VECTOR_SUPPORT` within the LPC2xxx variant HAL allows manual control of this facility.

If the application does not rely on a ROM monitor for diagnostic services then by default serial port UART0 will be claimed for HAL diagnostics. If using the Prototype Board, it becomes possible to use UART1 as well, and in order to allow its selection as a potential debug or diagnostic channel, the number of communications channels can be increased from 1 to 2 with the option `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS`. To specify which serial port is used for diagnostics, the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` may be set accordingly, and its baud rate configured with `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multiplier may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation. Note there are frequency constraints on the Current Controlled Oscillator (CCO) within the LPC2xxx, and the datasheet should be consulted to ensure the required specifications are not exceeded.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. To build eCos in Thumb mode, enable the `CYGHWR_THUMB` configuration option in the ARM architecture HAL.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. For example, allowing a Thumb application to be used with eCos built in normal ARM mode. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. eCos may be built with Thumb interworking support by enabling the `CYGBLD_ARM_ENABLE_THUMB_INTERWORK` CDL option in the ARM architecture HAL. Use of the LPC2xxx Flash driver requires Thumb interworking support to be enabled as the calls into the IAP firmware are must be made allowing a switch to Thumb mode.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the EA QuickStart board hardware, and should be read in conjunction with that specification. The QuickStart Board platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks), Memory Mapping control registers to map SRAM to 0x0, and Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. The size of this region depends on the LPC2xxx microcontroller variant in use. In the case of the LPC2148, the region is of size 512Kbytes, ending at 0x80000. However the last few blocks of Flash are reserved for use as bootblocks for the ISP/IAP firmware, resulting in a usable Flash size of 500Kbytes, ending at 0x7d000. The MAM is enabled to accelerate memory reads from this area.

internal SRAM

This is located at address 0x40000000 of the memory space, and is 16, 32 or 64k in size, depending on the chip fitted. The first 64 bytes are mapped to location 0x0000000. If using GDB stubs ROM, or another ROM monitor, the virtual vector table starts at 0x40000050 and extends to 0x40000150. The remainder of SRAM is available for use by applications. For RAM startup applications, SRAM below 0x40001000 is reserved for the GDB stubrom and the remainder is available for the application. An exception is if the on-chip Flash driver is to be used. In that case, the top 32 bytes of SRAM are used by it. This is automatically handled in the port's memory layout files if the flash driver is present in the configuration.

on-chip peripherals

These are accessible at location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2xxx User Manual for the appropriate microcontroller variant.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used   420 size  3920
Startup : Interrupt stack used   148 size  4096
Startup : Idlethread stack used    88 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

```
Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took  15.31 microseconds (9 raw clock ticks)
```

Testing parameters:

```
Clock samples:      32
Threads:            1
Thread switches:    128
Mutexes:            32
Mailboxes:          21
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence			
Ave	Min	Max	Var	Ave	Min	Function	
=====	=====	=====	=====	=====	=====	=====	
15.00	15.00	15.00	0.00	100%	100%	Create thread	
5.00	5.00	5.00	0.00	100%	100%	Yield thread [all suspended]	
3.33	3.33	3.33	0.00	100%	100%	Suspend [suspended] thread	
5.00	5.00	5.00	0.00	100%	100%	Resume thread	
6.67	6.67	6.67	0.00	100%	100%	Set priority	
1.67	1.67	1.67	0.00	100%	100%	Get priority	
11.67	11.67	11.67	0.00	100%	100%	Kill [suspended] thread	
3.33	3.33	3.33	0.00	100%	100%	Yield [no other] thread	
6.67	6.67	6.67	0.00	100%	100%	Resume [suspended low prio] thread	
5.00	5.00	5.00	0.00	100%	100%	Resume [runnable low prio] thread	
6.67	6.67	6.67	0.00	100%	100%	Suspend [runnable] thread	
5.00	5.00	5.00	0.00	100%	100%	Yield [only low prio] thread	
5.00	5.00	5.00	0.00	100%	100%	Suspend [runnable->not runnable]	
11.67	11.67	11.67	0.00	100%	100%	Kill [runnable] thread	
8.33	8.33	8.33	0.00	100%	100%	Destroy [dead] thread	

16.67	16.67	16.67	0.00	100%	100%	Destroy [runnable] thread
25.00	25.00	25.00	0.00	100%	100%	Resume [high priority] thread
1.41	0.00	1.67	0.44	84%	15%	Scheduler lock
3.49	3.33	5.00	0.28	90%	90%	Scheduler unlock [0 threads]
3.49	3.33	5.00	0.28	90%	90%	Scheduler unlock [1 suspended]
3.41	3.33	5.00	0.15	95%	95%	Scheduler unlock [many suspended]
3.52	3.33	5.00	0.32	89%	89%	Scheduler unlock [many low prio]
1.82	1.67	3.33	0.28	90%	90%	Init mutex
4.38	3.33	5.00	0.78	62%	37%	Lock [unlocked] mutex
5.00	5.00	5.00	0.00	100%	100%	Unlock [locked] mutex
4.11	3.33	5.00	0.83	53%	53%	Trylock [unlocked] mutex
3.80	3.33	5.00	0.67	71%	71%	Trylock [locked] mutex
1.35	0.00	1.67	0.51	81%	18%	Destroy mutex
21.67	21.67	21.67	0.00	100%	100%	Unlock/Lock mutex
1.98	1.67	3.33	0.51	80%	80%	Create mbox
1.19	0.00	1.67	0.68	71%	28%	Peek [empty] mbox
4.60	3.33	5.00	0.61	76%	23%	Put [first] mbox
1.43	0.00	1.67	0.41	85%	14%	Peek [1 msg] mbox
4.52	3.33	5.00	0.68	71%	28%	Put [second] mbox
0.87	0.00	1.67	0.83	52%	47%	Peek [2 msgs] mbox
4.76	3.33	5.00	0.41	85%	14%	Get [first] mbox
4.76	3.33	5.00	0.41	85%	14%	Get [second] mbox
4.05	3.33	5.00	0.82	57%	57%	Tryput [first] mbox
3.89	3.33	5.00	0.74	66%	66%	Peek item [non-empty] mbox
4.29	3.33	5.00	0.82	57%	42%	Tryget [non-empty] mbox
3.81	3.33	5.00	0.68	71%	71%	Peek item [empty] mbox
4.05	3.33	5.00	0.82	57%	57%	Tryget [empty] mbox
1.35	0.00	1.67	0.51	80%	19%	Waiting to get mbox
1.43	0.00	1.67	0.41	85%	14%	Waiting to put mbox
2.30	1.67	3.33	0.79	61%	61%	Delete mbox
15.00	15.00	15.00	0.00	100%	100%	Put/Get mbox
1.72	1.67	3.33	0.10	96%	96%	Init semaphore
3.91	3.33	5.00	0.75	65%	65%	Post [0] semaphore
3.96	3.33	5.00	0.78	62%	62%	Wait [1] semaphore
3.91	3.33	5.00	0.75	65%	65%	Trywait [0] semaphore
3.75	3.33	5.00	0.63	75%	75%	Trywait [1] semaphore
1.72	1.67	3.33	0.10	96%	96%	Peek semaphore
1.46	0.00	1.67	0.36	87%	12%	Destroy semaphore
13.91	13.33	15.00	0.75	65%	65%	Post/Wait semaphore
1.93	1.67	3.33	0.44	84%	84%	Create counter
1.46	0.00	1.67	0.36	87%	12%	Get counter value
1.35	0.00	1.67	0.51	81%	18%	Set counter value
4.32	3.33	5.00	0.80	59%	40%	Tick counter
1.46	0.00	1.67	0.36	87%	12%	Delete counter
1.72	1.67	3.33	0.10	96%	96%	Init flag
4.06	3.33	5.00	0.82	56%	56%	Destroy flag
3.59	3.33	5.00	0.44	84%	84%	Mask bits in flag
4.06	3.33	5.00	0.82	56%	56%	Set bits in flag [no waiters]
5.63	5.00	6.67	0.78	62%	62%	Wait for flag [AND]

```

5.63    5.00    6.67    0.78    62%  62% Wait for flag [OR]
5.57    5.00    6.67    0.75    65%  65% Wait for flag [AND/CLR]
5.52    5.00    6.67    0.72    68%  68% Wait for flag [OR/CLR]
1.35    0.00    1.67    0.51    81%  18% Peek on flag

2.86    1.67    3.33    0.67    71%  28% Create alarm
6.61    5.00    6.67    0.10    96%   3% Initialize alarm
3.91    3.33    5.00    0.75    65%  65% Disable alarm
6.15    5.00    6.67    0.72    68%  31% Enable alarm
4.38    3.33    5.00    0.78    62%  37% Delete alarm
4.95    3.33    5.00    0.10    96%   3% Tick counter [1 alarm]
23.33   23.33   23.33    0.00   100% 100% Tick counter [many alarms]
8.28    6.67    8.33    0.10    96%   3% Tick & fire counter [1 alarm]
138.96  138.33  140.00    0.78    62%  62% Tick & fire counters [>1 together]
26.88   26.67   28.33    0.36    87%  87% Tick & fire counters [>1 separately]
15.00   15.00   15.00    0.00   100% 100% Alarm latency [0 threads]
15.00   15.00   15.00    0.00   100% 100% Alarm latency [many threads]
26.69   26.67   30.00    0.05    99%  99% Alarm -> thread resume latency

3.38    3.33    5.00    0.00                    Clock/interrupt latency

8.33    8.33    8.33    0.00                    Clock DSR latency

312     312     312  (main stack:  716) Thread stack used (1360 total)
      All done, main stack : stack used   716 size  3920
      All done : Interrupt stack used    204 size  4096
      All done : Idlethread stack used    256 size  2048

```

Timing complete - 23650 ms total

PASS:<Basic timing OK>

EXIT:<done>

LED use

LEDs are available on the Prototype board. Most of these are attached to lines associated with peripherals present on the Prototype board, or the QuickStart board itself. However 9 LEDs are available for application use from C. The following C function may be used:

```

#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);

```

Values from 0 to 511 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off. The LEDs used are connected to P0.10-P0.13 and P0.17-P0.21, and with P0.21 being the MSB, and P0.10 the LSB.

The LEDs are also used during platform initialization and only P0.10 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

Other Issues

The following pin assignments are configured by default for LPC2148 at board initialisation time:

PINSEL0:

- P0.0/P0.1 for UART0
- P0.2/P0.3 for I2C
- P0.4/P0.5/P0.6 for SPI
- P0.7 as PWM2 for buzzer on Prototype board
- P0.8/P0.9 for UART1 (which is available on prototype board)
- P0.10-P0.13 as GPIO-controlled LEDs
- P0.14 EINT1 (available as button on prototype board)
- P0.15 EINT2 (available as button on prototype board)

PINSEL1:

- P0.16 EINT0 (available as button on prototype board)
 - P0.17-P0.21 as GPIO-controlled LEDs
 - P0.22 as GPIO output for SPI_SEL_MMC on prototype board
 - P0.23 as GPIO output for SPI_SEL_LED on prototype board
 - P0.29 as GPIO input for MMC/SD card detect on prototype board
 - P0.30 as EINT3 (available as button on prototype board)
(rather than GPIO input for SD write protect on prototype board)
- All other pins set as GPIO inputs

PINSEL2:

- P1.26-P1.31 for JTAG
- All other pins set as inputs

XIV. IAR KickStart Card Support

Overview

Name

eCos Support for the IAR KickStart Cards — Overview

Introduction

This platform HAL is designed to support the KickStart Card series from IAR, fitted with an NXP LPC2xxx microcontroller.

The support for the KickStart board series provided by this HAL has been initially developed for the IAR LPC2106 KickStart Card. This HAL documentation therefore presently corresponds to that particular board instance, and future supported variants will cause this documentation to be updated accordingly.

Description

The IAR KickStart Board is fitted with an NXP LPC2xxx processor rated at up to 60MHz, which contains up to 64KB of SRAM and up to 512KB of FLASH, depending on choice of LPC2xxx variant. The board itself has two 9-pin RS-232 serial interfaces connected to the LPC2xxx on-chip UART 0 and UART 1, push buttons connected to interrupt lines, LEDs, a JTAG debug port, and a prototyping area. Refer to IAR's KickStart board documentation for full details.

The typical mode of operation for eCos development usually depends on the amount of memory available. On LPC2xxx variants with 64Kbytes or more of SRAM, a GDB stub ROM image is programmed into the LPC2xxx on-chip flash memory, and the board will boot this image from reset. While RedBoot may also be used, its larger RAM footprint requirements usually make it unsuitable. Both RedBoot and the GDB stub ROM provide GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using UART 0.

On LPC2xxx variants with less than 64Kbytes SRAM, it is typically expected that standalone applications will be programmed directly to on-chip Flash, either using a hardware JTAG/ICE unit, or by serial using the on-chip In-System Programming (ISP) mechanism included with NXP LPC2xxx microcontrollers and a suitable host application running on a PC.

This documentation describes platform-specific elements of the IAR KickStart Board support within eCos. Documentation on the [NXP LPC2xxx variants](#) is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The NXP LPC2xxx microcontrollers on the IAR KickStart Boards have up to 512Kbytes of on-chip Flash memory. In a typical setup, the GDB stub ROM or the user application will load and run from this internal flash. For initial programming of the internal Flash, external support is required, such as the NXP LPC2000 Flash Utility, the Flash Magic tool, or a hardware JTAG/ICE unit. The latter may be used with its own in-built LPC2xxx flash programming support if it exists, or the eCosPro® **ecoflash** utility. 28Kbytes of internal flash memory should be reserved for the

GDB Stub ROM, the remainder being free for the application's use. Note that the LPC2xxx primary boot loader and IAP code reside in boot blocks located at the end of on-chip Flash. To determine the number and size of blocks reserved for their use, consult the specific LPC2xxx variant's datasheet.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2xxx memory mapping control to location 0x00000000. When loading applications using the GDB stub ROM, SRAM from location 0x40000040 to 0x40001000 is reserved for its use. The rest of SRAM is generally available for use by the application. Programs booted from ROM, or loaded directly into SRAM via JTAG may use all SRAM. In all cases, if using the eCos LPC2xxx Flash driver, the last 32 bytes (or more if a separate program buffer is used) become reserved due to the requirements of the IAP code.

The NXP LPC2xxx variant HAL includes support for the two LPC2xxx on-chip serial devices and is [documented in the variant HAL](#). Although the interrupt-driven serial driver supports the line status and modem control lines on UART 1 (UART 0 not having such support), the KickStart boards do not connect these pins, and so that functionality is unavailable.

The IAR KickStart port includes support for the on-chip watchdog, RTC (wallclock), interrupt controller (VIC) and on-chip Flash. This support is documented in the [LPC2xxx variant HAL](#).

Tools

The KickStart Board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was created using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Setup

Name

Setup — Preparing the IAR KickStart Board for eCos Development

Overview

In a typical development environment, the IAR KickStart Board boots from internal flash into either the GDB stub ROM monitor or directly into the user application. In the case of microcontrollers with less than 64Kbytes of SRAM, the latter is recommended. eCos applications to be loaded and run from the GDB stub ROM monitor may be configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable ROM image into Flash memory, either the GDB stub ROM or application images.

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the application, or if applicable, GDB stub ROM. A "straight through" 9-pin RS232 serial cable, with Male<->Female connectors is required. Using a "null modem" serial cable will not work.

Initial Installation

Board setup

Jumper settings must be checked and potentially changed on the board to ensure correct operation. This section describes jumper settings that are known to require attention. In general, any board-specific documentation from IAR takes precedence over the documentation here, as this may reflect hardware which has been modified since the time of writing of this documentation. Many KickStart boards are similar to each other, the only change of note being of course the choice of LPC2xxx microcontroller fitted. But if your board does not fit the description here (which has initially been based on the LPC2106 KickStart) then you should consult the board documentation.

Firstly, there are two jumpers located close to the serial connectors, labelled JP3 and JP4 on the LPC2106 KickStart. These can both be closed (i.e. jumper fitted and connecting the two pins) in order to permit use of both serial ports.

There are two jumpers labelled EN_SW_ISP and EN_SW_RST close to the push buttons, also labelled JP7 and JP8 respectively on the LPC2106 KickStart. In general, these jumpers should only be closed when wishing to reprogram the on-chip Flash via ISP. Otherwise they should remain open (jumper not connecting the two pins) so that any unplugging of the serial connector, movement of the serial connector, or use of flow control signals from the host PC, do not cause a spurious reset or interrupt (on the EINT1 line) of the board.

The jumpers controlling the LEDs (labelled LED Jumper Block on the LPC2106 KickStart) may remain in their default state of being connected to P0.0-P0.15. This is assumed by code which allows use of the user-configurable LEDs, as described [later](#).

All other jumpers can remain in their factory-supplied default state.

Flash Installation

This process assumes that a Microsoft Windows machine with the Flash Magic utility installed is available. Flash Magic is a tool for programming flash based microcontrollers from NXP using a protocol via the RS232 serial port to communicate with the In-System Programming (ISP) firmware on the LPC2xxx. The Flash Magic utility is sponsored by NXP and available from this website (<http://www.flashmagictool.com/>).

The first step is to connect the RS232 cable between UART0 of the KickStart board and the host PC. Do not use UART1. Now close the two jumpers mentioned earlier labelled EN_SW_ISP and EN_SW_RST. These allow the software on the PC to reset the LPC2xxx and enter the ISP firmware. Finally apply the power.

Start the Flash Magic utility on the host PC, and a window will be displayed allowing various parameters to be configured in a series of steps. For step 1, firstly choose the appropriate COM port that is being used on your PC and set the Baud Rate to 38400 baud. Next select the appropriate LPC2xxx device in use such as LPC2106. The "Interface" should be set to "None (ISP)". And finally for step 1 choose the appropriate Oscillator Frequency for the KickStart board in use. This may be found in the board documentation, and is usually visibly readable on the surface of the oscillator on the board (in a metal package). For example for the LPC2106 KickStart, the oscillator reads T14.7456 indicating 14.7456MHz.

For step 2, it is usually adequate to leave the option "Erase blocks used by hex file" checked, and ignore the other settings. For step 3, you must select the program image to be downloaded, in Intel HEX format. To program the pre-built GDB stub ROM image, locate the file `gdb_module.hex` in the `loaders` subdirectory of your release. To generate an Intel HEX format version of an application you have built yourself run the following command at a shell prompt:

```
$ arm-elf-objcopy -O ihex app.elf app.hex
```

This converts the application image in ELF format (as output by the linker), to Intel HEX format in the file `app.hex`. Note that the arm-elf tools must be on your path at this point. If they are not, run the command:

```
$ . /opt/ecos/ecosenv.sh
```

before you run the above **arm-elf-objcopy** command.

In step 4, it is recommended to set the option "Verify after programming". Finally it is possible to click on "Start" to program the image into the on-chip Flash.

Tip: If there is a problem communicating with the board, such as a report of a failure to autobaud, then this may imply that the Flash Magic tool was not able to control the serial lines properly. This can happen with some USB-Serial converters. To workaround this issue, power the board off and remove (i.e. open) the EN_SW_ISP and EN_SW_RST jumpers. Then simultaneously press the buttons marked 'Reset' and 'ISP/INT1' then release the Reset button, and finally release the 'ISP/INT1' button. This is an alternative mechanism of forcing the ISP firmware to be entered. Once this has been successfully performed, the programming operation may be retried from the Flash Magic utility.

When the process completes, remove (i.e. open) the two jumpers next to the serial port. If a GDB stub ROM image has been programmed, verify the programming has been successful by starting a terminal emulation application such as HyperTerminal on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Reset the board and the stubrom should start. For boards programmed with GDB stubs the output should be similar to the following:

```
+$T050f:ec070000;0d:28080040;#52
```

This is the stubrom attempting to communicate with GDB and indicates that it is functioning correctly.

Rebuilding the GDB Stub ROM

Should it prove necessary to rebuild the GDB Stub ROM binary, this is done most conveniently at the command line. Your PATH and ECOS_REPOSITORY environment variables must first be set correctly. The following steps given an example of how to rebuild the stubs for a KickStart board with LPC2106:

```
$ mkdir stub_rom
$ cd stub_rom
$ ecosconfig new iar_kickstart_lpc2106 stubs
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the files `gdb_module.img` (ELF format), `gdb_module.srec` (Motorola S-Record format), `gdb_module.bin` (raw binary format), and `gdb_module.hex` (Intel HEX format).

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The IAR KickStart platform HAL package is loaded automatically when eCos is configured for a target which uses it. The target names include the LPC2xxx model in use. At this time the only target supported is the `iar_kickstart_lpc2106` target. It should never be necessary to load this platform HAL package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The IAR KickStart platform HAL package supports three separate startup types:

ROM

This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. On targets with less than 64Kbytes of SRAM, this is the startup type normally used.

RAM

This is the startup type which is normally used during application development on targets with 64Kbytes of SRAM or greater. The board has the GDB stubrom or RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the ROM monitor. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the ROM monitor, including diagnostic output. Larger applications may not fit into the available SRAM, in which case ROM startup may be required.

JTAG

This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading. Some sample configuration and initialisation scripts for a number of JTAG debugging solutions may be found in the `misc` subdirectory of the platform HAL package within the component repository.

The ROM Monitor and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubrom or RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup. Virtual vector support is usually only required for RAM startup applications or ROM monitors. The CDL option `CYGFUN_HAL_LPC2XXX_VIRTUAL_VECTOR_SUPPORT` within the LPC2xxx variant HAL allows manual control of this facility.

If the application does not rely on a ROM monitor for diagnostic services then by default serial port UART0 will be claimed for HAL diagnostics. If using the Prototype Board, it becomes possible to use UART1 as well, and in order to allow its selection as a potential debug or diagnostic channel, the number of communications channels can be increased from 1 to 2 with the option `CYGNUM_HAL_VIRTUAL_VECTOR_COMM_CHANNELS`. To specify which serial port is used for diagnostics, the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` may be set accordingly, and its baud rate configured with `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD`.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multiplier may be configured to allow a core clock (CCLK) speed of up to 60MHz. The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation. Note there are frequency constraints on the Current Controlled Oscillator (CCO) within the LPC2xxx, and the datasheet should be consulted to ensure the required specifications are not exceeded.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. To build eCos in Thumb mode, enable the `CYGHWR_THUMB` configuration option in the ARM architecture HAL.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. For example, allowing a Thumb application to be used with eCos built in normal ARM mode. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. eCos may be built with Thumb interworking support by enabling the `CYGBLD_ARM_ENABLE_THUMB_INTERWORK` CDL option in the ARM architecture HAL. Use of the LPC2xxx Flash driver requires Thumb interworking support to be enabled as the calls into the IAP firmware are must be made allowing a switch to Thumb mode.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the IAR KickStart board hardware, and should be read in conjunction with that specification. The KickStart Board platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. This includes the PINSEL functions and LED bank. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks), Memory Mapping control registers to map SRAM to 0x0, and Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. The size of this region depends on the LPC2xxx microcontroller variant in use. In the case of the LPC2106, the region is of size 128Kbytes, ending at 0x20000. However the last few blocks of Flash are reserved for use as bootblocks for the ISP/IAP firmware, resulting in a usable Flash size of 120Kbytes, ending at 0x1e000. The MAM is enabled to accelerate memory reads from this area.

internal SRAM

This is located at address 0x40000000 of the memory space, and is 16, 32 or 64k in size, depending on the chip fitted. The first 64 bytes are mapped to location 0x0000000. If using GDB stubs ROM, or another ROM monitor, the virtual vector table starts at 0x40000050 and extends to 0x40000150. The remainder of SRAM is available for use by applications. For RAM startup applications, SRAM below 0x40001000 is reserved for the GDB stubrom and the remainder is available for the application. An exception is if the on-chip Flash driver is to be used. In that case, the top 32 bytes of SRAM are used by it. This is automatically handled in the port's memory layout files if the flash driver is present in the configuration.

on-chip peripherals

These are accessible at location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2xxx User Manual for the appropriate microcontroller variant.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used   420 size  3920
Startup : Interrupt stack used   148 size  4096
Startup : Idlethread stack used    80 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

```
Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took  14.94 microseconds (8 raw clock ticks)
```

Testing parameters:

```
Clock samples:      32
Threads:            2
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence			
Ave	Min	Max	Var	Ave	Min	Function	
=====	=====	=====	=====	=====	=====	=====	
13.56	13.56	13.56	0.00	100%	100%	Create thread	
3.39	3.39	3.39	0.00	100%	100%	Yield thread [all suspended]	
4.24	3.39	5.09	0.85	100%	50%	Suspend [suspended] thread	
3.39	3.39	3.39	0.00	100%	100%	Resume thread	
5.09	5.09	5.09	0.00	100%	100%	Set priority	
1.70	1.70	1.70	0.00	100%	100%	Get priority	
11.87	11.87	11.87	0.00	100%	100%	Kill [suspended] thread	
4.24	3.39	5.09	0.85	100%	50%	Yield [no other] thread	
5.93	5.09	6.78	0.85	100%	50%	Resume [suspended low prio] thread	
4.24	3.39	5.09	0.85	100%	50%	Resume [runnable low prio] thread	
5.93	5.09	6.78	0.85	100%	50%	Suspend [runnable] thread	
3.39	3.39	3.39	0.00	100%	100%	Yield [only low prio] thread	
3.39	3.39	3.39	0.00	100%	100%	Suspend [runnable->not runnable]	
11.87	11.87	11.87	0.00	100%	100%	Kill [runnable] thread	
8.48	8.48	8.48	0.00	100%	100%	Destroy [dead] thread	

15.26	15.26	15.26	0.00	100%	100%	Destroy [runnable] thread
22.04	20.35	23.74	1.70	100%	50%	Resume [high priority] thread
8.38	6.78	11.87	0.23	92%	7%	Thread switch
1.38	0.00	1.70	0.52	81%	18%	Scheduler lock
3.31	1.70	3.39	0.15	95%	4%	Scheduler unlock [0 threads]
3.31	1.70	3.39	0.15	95%	4%	Scheduler unlock [1 suspended]
3.31	1.70	3.39	0.15	95%	4%	Scheduler unlock [many suspended]
3.31	1.70	3.39	0.15	95%	4%	Scheduler unlock [many low prio]
1.75	1.70	3.39	0.10	96%	96%	Init mutex
4.24	3.39	5.09	0.85	100%	50%	Lock [unlocked] mutex
4.82	3.39	5.09	0.45	84%	15%	Unlock [locked] mutex
4.08	3.39	5.09	0.82	59%	59%	Trylock [unlocked] mutex
3.71	3.39	5.09	0.52	81%	81%	Trylock [locked] mutex
1.38	0.00	1.70	0.52	81%	18%	Destroy mutex
20.35	20.35	20.35	0.00	100%	100%	Unlock/Lock mutex
1.96	1.70	3.39	0.45	84%	84%	Create mbox
1.38	0.00	1.70	0.52	81%	18%	Peek [empty] mbox
4.50	3.39	5.09	0.76	65%	34%	Put [first] mbox
1.27	0.00	1.70	0.64	75%	25%	Peek [1 msg] mbox
4.56	3.39	5.09	0.73	68%	31%	Put [second] mbox
1.22	0.00	1.70	0.68	71%	28%	Peek [2 msgs] mbox
4.56	3.39	5.09	0.73	68%	31%	Get [first] mbox
4.56	3.39	5.09	0.73	68%	31%	Get [second] mbox
3.97	3.39	5.09	0.76	65%	65%	Tryput [first] mbox
3.39	3.39	3.39	0.00	100%	100%	Peek item [non-empty] mbox
4.40	3.39	5.09	0.82	59%	40%	Tryget [non-empty] mbox
3.66	3.39	5.09	0.45	84%	84%	Peek item [empty] mbox
3.92	3.39	5.09	0.73	68%	68%	Tryget [empty] mbox
1.38	0.00	1.70	0.52	81%	18%	Waiting to get mbox
1.43	0.00	1.70	0.45	84%	15%	Waiting to put mbox
2.44	1.70	3.39	0.83	56%	56%	Delete mbox
13.88	13.56	15.26	0.52	81%	81%	Put/Get mbox
1.70	1.70	1.70	0.00	100%	100%	Init semaphore
3.71	3.39	5.09	0.52	81%	81%	Post [0] semaphore
3.97	3.39	5.09	0.76	65%	65%	Wait [1] semaphore
3.66	3.39	5.09	0.45	84%	84%	Trywait [0] semaphore
3.60	3.39	5.09	0.37	87%	87%	Trywait [1] semaphore
1.75	1.70	3.39	0.10	96%	96%	Peek semaphore
1.70	1.70	1.70	0.00	100%	100%	Destroy semaphore
13.56	13.56	13.56	0.00	100%	100%	Post/Wait semaphore
2.01	1.70	3.39	0.52	81%	81%	Create counter
1.70	1.70	1.70	0.00	100%	100%	Get counter value
1.22	0.00	1.70	0.68	71%	28%	Set counter value
4.13	3.39	5.09	0.83	56%	56%	Tick counter
1.43	0.00	1.70	0.45	84%	15%	Delete counter
1.70	1.70	1.70	0.00	100%	100%	Init flag
4.13	3.39	5.09	0.83	56%	56%	Destroy flag
3.50	3.39	5.09	0.20	93%	93%	Mask bits in flag
4.03	3.39	5.09	0.79	62%	62%	Set bits in flag [no waiters]

```

5.40    5.09    6.78    0.52    81%    81% Wait for flag [AND]
5.30    5.09    6.78    0.37    87%    87% Wait for flag [OR]
5.30    5.09    6.78    0.37    87%    87% Wait for flag [AND/CLR]
5.35    5.09    6.78    0.45    84%    84% Wait for flag [OR/CLR]
1.22    0.00    1.70    0.68    71%    28% Peek on flag

2.70    1.70    3.39    0.82    59%    40% Create alarm
6.46    5.09    6.78    0.52    81%    18% Initialize alarm
3.81    3.39    5.09    0.64    75%    75% Disable alarm
6.04    5.09    6.78    0.83    56%    43% Enable alarm
4.29    3.39    5.09    0.84    53%    46% Delete alarm
4.82    3.39    5.09    0.45    84%    15% Tick counter [1 alarm]
22.15   22.04   23.74    0.20    93%    93% Tick counter [many alarms]
8.05    6.78    8.48    0.64    75%    25% Tick & fire counter [1 alarm]
138.82  137.33  139.03    0.37    87%    12% Tick & fire counters [>1 together]
25.70   25.43   27.13    0.45    84%    84% Tick & fire counters [>1 separately]
13.56   13.56   13.56    0.00   100%   100% Alarm latency [0 threads]
15.54   13.56   18.65    1.49    50%    32% Alarm latency [2 threads]
15.54   13.56   18.65    1.49    50%    32% Alarm latency [many threads]
27.14   27.13   28.82    0.02    99%    99% Alarm -> thread resume latency

3.39    3.39    3.39    0.00                    Clock/interrupt latency

7.93    6.78   11.87    0.00                    Clock DSR latency

272     272     272 (main stack: 764) Thread stack used (1360 total)
      All done, main stack : stack used 764 size 3920
      All done : Interrupt stack used 204 size 4096
      All done : Idlethread stack used 248 size 2048

```

Timing complete - 30220 ms total

PASS:<Basic timing OK>

EXIT:<done>

LED use

LEDs are available on the KickStart boards although most of these are attached to lines associated with peripherals. However 4 LEDs are available for application use from C. The following C function may be used:

```

#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);

```

Values from 0 to 15 will be displayed on the LED bank representing the binary value with 1 being on and 0 being off. The LEDs used are connected to P0.10-P0.13 P0.13 being the MSB, and P0.10 the LSB.

The LEDs are also used during platform initialization and only P0.10 should be illuminated if booting has been successful. Other LED indications represent the stage in the initialization process that failed.

Other Issues

The following pin assignments are configured by default for LPC2106 at board initialisation time:

PINSEL0:

P0.0/P0.1 for UART0

P0.2/P0.3 for I2C

P0.4/P0.5/P0.6/P0.7 for SPI

P0.8/P0.9 for UART1

P0.10-P0.13 as GPIO-controlled LEDs

P0.14 EINT1

P0.15 EINT2

PINSEL1:

P0.16 EINT0

P0.17-P0.21 as GPIO, although in practice these are used for JTAG
if a JTAG unit is connected.

P0.22-P0.31 as GPIO inputs

XV. Embedded Artists LPC2468 OEM Board Support

Overview

Name

eCos Support for the Embedded Artists LPC2468 OEM Board — Overview

Description

The Embedded Artists LPC2468 OEM Board is fitted with an NXP LPC2468 processor rated up to 72MHz, which contains 64KB of SRAM and 512KB of FLASH. When used in conjunction with the OEM base board, it provides access to two on-chip UARTs (one via USB, one via a 9-pin connector), a single GPIO LED, an MMC/SD card socket, and a PHY connected to the on-chip Ethernet MAC. Refer to the board documentation for full details.

Two variants of the LPC2468 OEM board are supported. The LPC2468-16 board provides a 16 bit data bus to external RAM, works with all versions of the base board, and supports UART 1. The LPC2468-32 board provides a 32 bit data bus to external RAM, works only with v1.4 and later versions of the base board, and does not allow UART 1 to be used.

For typical eCos development, a RedBoot image is programmed into the LPC2468 on-chip flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using either UART or via the ethernet.

This documentation describes platform-specific elements of the LPC2468 OEM Board support within eCos. Documentation on the [NXP LPC2xxx variants](#) is available separately, and should be read in conjunction with this documentation. The LPC2xxx documentation covers various topics including HAL support common to LPC2xxx variants, and on-chip device support. This document complements the LPC2xxx documentation.

Supported Hardware

The LPC2468 OEM Board has 512Kbyte of on-chip Flash memory. In a typical setup, RedBoot will load and run from this internal flash. An initial image must be programmed into this flash using either the FlashMagic utility, or via a JTAG debugger. Following this, it may be reprogrammed using flash drivers in RedBoot.

The first 64 bytes of on-chip SRAM are mapped by the HAL startup code using the LPC2468 memory mapping control to location 0x00000000 for speed of interrupt vector processing. The rest of SRAM is available for use by the application. 4MB of external NOR flash is available at 0x80000000; the topmost 64K of this is used by RedBoot for configuration data, the rest is available for application use and can be managed by RedBoot's flash file system. 32 MB of SDRAM is available at 0x81000000; the first 1MByte of this is reserved for use by RedBoot, the rest is available for the code and data of loaded applications.

The NXP LPC2xxx variant HAL includes support for the on-chip serial devices which is [documented in the variant HAL](#). The interrupt-driven serial driver supports the line status and modem control (including hardware handshaking) lines on UART1 only.

The LPC2468 OEM Board port includes support for the on-chip watchdog, RTC (wallclock), and interrupt controller (VIC). This support is documented in the [LPC2xxx variant HAL](#).

The on-chip Ethernet MAC is supported. The LPC2468-16 and LPC2468-32 use different PHYs, and both of these are supported.

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the OEM board.

Tools

The LPC2468 OEM Board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Setup

Name

Setup — Preparing the LPC2468 OEM Board for eCos Development

Overview

In a typical development environment, the LPC2468 OEM Board boots from internal flash into RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.hex
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.srec

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the RedBoot **baud** command.

Initial Installation

Flash Installation

This process assumes that a Microsoft Windows machine with the Embedded Systems Academy Flash Magic utility is available.

The first step is to set up the board as described in the Embedded Artists documentation. Install the FTDI USB driver from <http://www.ftdichip.com/Drivers/VCP.htm> and configure it as described in the Embedded Artists documentation. Install Flash Magic from <http://www.flashmagictool.com>.

Install the ISP jumpers (P2.10 and RESET) and press the reset button. The board is now running a special NXP boot loader. Start Flash Magic and set the Communications section to select the FTDI USB device, 38400 baud, device LPC2468, Interface “None (ISP)” and 12MHz Oscillator Frequency. Test communication with the board by using the “ISP->Read Device Signature” menu entry. If communication is not successful, check that the USB cable is connected, the ISP jumpers are installed and the correct COM port is being used.

Check “Erase blocks used by Hex File” under “Erase”. In the “Hex File” section, select the `redboot_ROM.hex` file. Under “Options”, all boxes should be clear except “Verify after programming”. Now press the “Start” button. The utility should show the progress of the upload.

When the process completes, the utility should be closed. Verify the programming has been successful by starting a terminal emulation application such as HyperTerminal or minicom on the host PC and set the serial communication parameters to 38400 baud, 8 data bits, no parity, 1 stop bit (8N1) and no flow control (handshaking). Remove the ISP jumpers. Reset the board and RedBoot should start. The output should be similar to the following:

```
***Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.8/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 17:48:38, Mar 10 2008

Platform: Embedded Artists LPC2468 OEM Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0xa0000000-0xa2000000, [0xa000ac08-0xalfed000] available
FLASH: 0x00000000-0x0007dfff, 8 x 0x1000 blocks, 14 x 0x8000 blocks, 6 x 0x1000s
FLASH: 0x80000000-0x803fffff, 64 x 0x10000 blocks
RedBoot>
```

It is now necessary to initialize the flash file system and the flash configuration. This can be done with the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x803f0000-0x803fffff: .
... Program from 0xalf0000-0xa2000000 to 0x803f0000: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.1.2
Console baud rate: 38400
DNS server IP address: 10.0.0.1
Network hardware address [MAC] for eth0: 0x0E:0x00:0x00:0xEA:0x18:0xF0
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: lpc2xxx
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x803f0000-0x803fffff: .
... Program from 0xalf0000-0xa2000000 to 0x803f0000: .
RedBoot>
```

Replace the IP addresses in the above with those for your own network. The above also accepts the default for the MAC address, if more than one LPC2468 is to be used on the same network then different MAC addresses should be used; Embedded Artists boards are supplied with a sticker showing an assigned MAC address, and this should be used by preference.

It is it ever necessary to reinstall RedBoot, the above directions can be repeated. Alternatively, a new RedBoot may be installed from RedBoot itself. It is not possible to do this directly, since RedBoot is executing from the flash that needs to be erased and reprogrammed. Instead it is necessary to run a RAM version of RedBoot, use that to download the new ROM RedBoot to RAM, and then program that to flash.

The following shows an example session to do this. It assumes that `redboot_RAM.srec` and `redboot_ROM.bin` are available via TFTP on the server set up in **fconfig**.

```
RedBoot> load redboot_RAM.srec
Using default protocol (TFTP)
Entry point: 0xa0100040, address range: 0xa0100000-0xa011bab4
RedBoot> go
+Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.8/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version UNKNOWN - built 11:00:52, Mar 11 2008

Platform: Embedded Artists LPC2468 OEM Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0xa0000000-0xa2000000, [0xa0125a28-0xa1fed000] available
FLASH: 0x00000000-0x0007dfff, 8 x 0x1000 blocks, 14 x 0x8000 blocks, 6 x 0x1000s
FLASH: 0x80000000-0x803fffff, 64 x 0x10000 blocks
RedBoot> load -r -b ${freememlo} redboot_ROM.bin
Using default protocol (TFTP)
Raw file loaded 0xa0125c00-0xa0142ba3, assumed entry at 0xa0125c00
RedBoot> fis write -f 0x00000000 -b ${freememlo} -l 0x20000
* CAUTION * about to program FLASH
          at 0x00000000..0x0001ffff from 0xa0125c00 - continue (y/n)? y
... Erase from 0x00000000-0x0001ffff: .....
... Program from 0xa0125c00-0xa0145c00 to 0x00000000: .....
RedBoot> reset
+Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.2.8/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 11:04:34, Mar 11 2008

Platform: Embedded Artists LPC2468 OEM Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited

RAM: 0xa0000000-0xa2000000, [0xa000ac08-0xa1fed000] available
FLASH: 0x00000000-0x0007dfff, 8 x 0x1000 blocks, 14 x 0x8000 blocks, 6 x 0x1000s
FLASH: 0x80000000-0x803fffff, 64 x 0x10000 blocks
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild the RedBoot binary, this is done most conveniently at the command line. Assuming your `PATH` and `ECOS_REPOSITORY` environment variables have been set correctly, the steps needed to rebuild RedBoot for the LPC2468-16 are:

Setup

```
$ mkdir redboot_ealpc2468_rom
$ cd redboot_ealpc2468_rom
$ ecosconfig new ea_lpc2468_16 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/lpc2xxx/ea_lpc2468/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.hex`.

Substitute 32 for 16 in the above to build RedBoot for the LPC2468-32 module.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The `ea_lpc2468` platform HAL package is loaded automatically when eCos is configured for an `ea_lpc2468_16` or `ea_lpc2468_16` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The `ea_lpc2468` platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash at location 0x0 in internal on-chip Flash and boots from that location. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into internal flash at location 0x0. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, or as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port UART0 will be claimed for HAL diagnostics.

Flash Driver

The LPC2468 OEM board contains an SST 39VF3201 NOR flash device. The `CYGPKG_DEVS_FLASH_SST_39VFXXX_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the Embedded Artists LPC2468 OEM board.

Ethernet Driver

The LPC2468 contains an ethernet MAC device. The `CYGPKG_DEVS_ETH_ARM_LPC2XXX` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the LPC2468 OEM board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The PLL multipliers and dividers may be configured to allow a core clock (CCLK) speed of up to 72MHz. However, the platform HAL currently sets the clock to 48MHz, duplicating the configuration in the supplied example code as a consequence of CPU errata affecting various revisions of the LPC2468. Setting the CPU revision with the `CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_CPU_REVISION` configuration option can be used to provide default clock settings appropriate to the CPU revision in use. If the CPU revision cannot be guaranteed it should be left as "Initial". The description of the clock-related CDL options may be found in the LPC2xxx variant HAL documentation.

I²C Bus Configuration

The on-chip I²C devices are supported by a driver in the variant HAL package. Each bus for this driver needs to be configured in the platform HAL with the following options:

`CYGPKG_HAL_ARM_LPC2XXX_I2CX`

This is the master component, enabling this activates all the other configuration options and causes the driver to create the data structures to access this bus.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_CLOCK`

Bus clock speed in Hz. Usually frequencies of either 100kHz or 400kHz are chosen, the latter sometimes known as fast mode.

`CYGPKG_HAL_ARM_LPC2XXX_I2CX_SDA`

This option describes the pin used for SDA on this bus. This takes the form of an invocation of the macro `__LPC2XXX_PINSEL_FUNC`. Parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2468 user manual for details of which pins may be used by each bus.

CYGPKG_HAL_ARM_LPC2XXX_I2CX_SCL

This option describes the pin used for SCL on this bus. Like SDA this takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

Note that "I2CX" is a placeholder for a given bus instance: "I2C0", "I2C1" or "I2C2". By default the platform HAL only enables I²C bus 0 in order to access the PCA9532 LED controller on the base board.

SPI Bus Configuration

The on-chip SSP SPI devices (not the Legacy SPI device) are supported by the NXPSSP driver package, CYGPKG_DEVS_SPI_ARM_NXPSSP. This needs some configuration in the platform HAL:

CYGPKG_HAL_ARM_LPC2XXX_SPI

This is the master component, enabling this activates all the other configuration options. It also causes `ea_lpc2468_spi.c` to be compiled, which contains descriptions of the devices on the SPI buses.

CYGPKG_HAL_ARM_LPC2XXX_SPIX

This is the master component for each bus. Enabling this activates the other configuration options for this bus, and causes the driver to support this bus.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_SCLK

This option describes the pin used for SCLK on SPIX. It takes the form of an invocation of `__LPC2XXX_PINSEL_FUNC`. The parameters are the port number, pin within that port, and the alternate select function for the pin. See the LPC2468 user manual for details."

CYGPKG_HAL_ARM_LPC2XXX_SPIX_MISO

This option describes the pin used for MISO on SPIX. Like SCLK it takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_MOSI

This option describes the pin used for MOSI on SPIX. Like SCLK it takes the form of a call to `__LPC2XXX_PINSEL_FUNC`.

CYGPKG_HAL_ARM_LPC2XXX_SPIX_CS_PINS

This defines the pins to be used as chip selects for this bus. It is a comma separated list of GPIO pin names, the first for device 0, the second for device 1, and so on. Pin names are defined in the `var_io.h` header in the LPC2xxx variant HAL.

Note that "SPIX" is a placeholder for a given bus instance: "SPI0" or "SPI1". By default the platform HAL only enables SPI0, for testing only.

MCI peripheral configuration

The on-chip Multimedia Card Interface (MCI) is supported to allow access to Multimedia Cards (MMC) or Secure Digital (SD) cards using the socket on the OEM board. This support is provided in conjunction

with the generic MMC/SD driver package (CYGPKG_DEVS_DISK_MMC), the Primecell MCI driver package (CYGPKG_DEVS_MMCS_D_ARM_PRIMECELL_MCI) and the LPC2xxx variant HAL in order to provide some elements of the DMA support. Documentation and configuration options within those packages should also be consulted. Note that the miniSD socket on the CPU board is not supported.

In order to configure the hardware for access to the socket, Jumper J47 on the base board must be set with pins 2-3 connected (P0.22 selected for MCIDAT0), and Jumper J27 must be set with MCIPWR active low.

The following CDL configuration options are used to control the behaviour of the MMC/SD card support:

MMC/SD card support (CYGPKG_HAL_ARM_LPC2XXX_EA_LPC2468_MCI)

This option allows the MMC/SD card support as a whole to be enabled or disabled, although the generic disk device driver package (CYGPKG_IO_DISK) must be loaded in order to enable the MMC/SD support.

Use on-chip USB memory for DMA (CYGSEM_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_USE_USB_MEM_FOR_DMA)

The LPC2468 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2468 have errata which decreases their achievable CPU clock frequency.

Using on-chip memory dedicated to USB helps reduce or remove these problems, depending on CPU frequency. Clearly this option must be disabled if the on-chip USB peripheral is to be used. It is also desirable to disable this option if the CPU frequency is high enough, in order to remove an extra copy on every data transfer, thus improving performance. The USB memory used is 512 bytes at the start of the USB memory space (0x7FD00000).

If this option is disabled and the DMA is not able to proceed quickly enough, this will be visible in the form of I/O errors. In that case, if it is not possible to enable this option it is recommended to adjust the CYGDAT_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_BUS_SPEED_LIMIT configuration option.

Lock AHB bus during DMA transfer (CYGSEM_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_DMA_LOCKS_AHB)

The AMBA Hardware Bus (AHB) is used to connect AMBA peripherals within the LPC2468, including the ARM core, DMA controller and memory controllers. When this option is enabled, the AHB is locked for the duration of MCI DMA transfer bursts. If another AMBA host needs to make a transfer it may be delayed as a result, which may not be desirable.

Disabling this option allows the AHB arbiter to permit other AHB hosts to perform transfers. Of course this may mean the MCI DMA transfers can in turn themselves get delayed, risking data overruns or underruns in MCI transfers, resulting in I/O errors during block reads or writes. This is particularly likely on processors running at slower clock speeds where there may already be difficulties with the DMA servicing data transfers quickly enough.

MMC/SD bus frequency limit (CYGNUM_HAL_ARM_LPC2XXX_EA_LPC2468_MCI_BUS_SPEED_LIMIT)

The LPC2468 cannot always keep up with the data transfer requirements, especially at slower CPU clock speeds. This is because the DMA controller runs at the speed of the CPU clock (CCLK) along with the fact that some LPC2468 have errata which decreases their achievable CPU clock frequency. The adjacent options to use on-chip USB memory and to lock the AHB bus can help prevent this, but sometimes they are insufficient

to prevent data overruns or underruns resulting in I/O errors during block reads or writes. In which case the only remaining recourse is to reduce the required data transfer rate between the MCI and the card.

This option can be used to impose an upper limit on the MMC/SD bus frequency. The value used in this option is measured in Hertz, and the use of 4-bit mode with SD cards is not a factor - this option provides the bus frequency, so a 4-bit bus will transfer four times the amount of data as a 1-bit bus in the same time period.

Note that this option provides a limit, and does not mean the card bus will operate at that frequency. The frequency is also governed by what the card will support, and the resolution of the clock used to derive the MMC/SD clock signal, and how it can be divided down.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

However there are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used.

Onboard NAND

The HAL port includes a low-level driver to access the on-board Samsung K9F1G08U08 NAND flash memory chip. To enable the driver, activate the CDL option `CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_NAND` and ensure that the `CYGPKG_DEVS_NAND_SAMSUNG_K9` package is present in your eCos configuration. The driver is capable of operating with or without the `NAND_RDY` line connected.

`CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_USE_NAND_RDY`

The EA OEM Base Board provides a jumper which connects the ready line of the NAND chip (`NAND_RDY`) to pin P2.12 on the CPU. Setting this option indicates to the driver that that jumper, or similar layout with the same effect, is in place. This provides an improvement in efficiency, but must not be set if the jumper is not so connected.

CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_NAND_RDY_USE_INTERRUPT

(Only active if CYGHWR_HAL_ARM_LPC2XXX_EA_LPC2468_USE_NAND_RDY is set.) If set, pin P2.12 (see above) is set up as an interrupt (EINT2). Setting this causes the thread invoking the driver to sleep when waiting for a program or erase operation to complete, as opposed to entering a polling loop. This potentially represents an efficiency gain if you have at least one other thread which can carry on performing useful work while the NAND chip works.

If this option is not set, the driver polls the ready line.

When this option is set, the driver automatically detects whether the eCos kernel scheduler is running; if it is not, interrupt mode cannot operate, and the driver falls back to polling the ready line.

Interrupt mode imposes its own overheads on the driver thread. Benchmarking chip program and erase operations alone will necessarily appear to show a slow-down in interrupt mode when the scheduler is running. This option can only improve efficiency on a holistic basis, and only then in the case where there are other threads which can continue to work while the driver is waiting for the NAND operation to complete.

Partitioning the NAND chip

The NAND chip must be partitioned before it can become available to applications.

A CDL script which allows the chip to be manually partitioned is provided (see CYGSEM_DEVS_NAND_EA_LPC2468_PARTITION_MANUAL_CONFIG); if you choose to use this, the relevant data structures will automatically be set up for you when the device is initialised. By default, the manual config CDL script sets up a single partition (number 0) encompassing the entire device.

It is possible to configure the partitions in some other way, should it be appropriate for your setup. To do so you will have to add appropriate code to `ea_lpc2468_nand.c`.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the LPC2468 OEM Board hardware, and should be read in conjunction with that specification. The LPC2468 platform HAL package complements the ARM architectural HAL and the LPC2xxx variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor to do most of this.

For ROM startup, the HAL will perform additional initialization, programming the various internal registers including PLL (for the clocks); Memory Mapping control registers to map SRAM to 0x0; the memory controller for access to external FLASH and SDRAM; and the Memory Acceleration Module (MAM). The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

on-chip Flash

This is located at address 0x0 of the memory space, although after hardware initialization, the start of internal SRAM is mapped over locations 0x0 to 0x40. This region ends at 0x80000. The MAM is enabled to accelerate memory reads from this area. A driver is available for using this flash via the eCos flash API.

external Flash

This is located at address 0x80000000 of the memory space. It is not used by default by eCos, although if RedBoot is asked to manage the Flash, it reserves flash addresses 0x803F0000 thru 0x803FEFFF. If RedBoot stores its configuration data in Flash, then addresses 0x803FF000 thru 0x803FFFFFF are reserved by RedBoot.

internal SRAM

This is located at address 0x40000000 of the memory space, ending at location 0x4000FFFF. The first 64 bytes are mapped to location 0x00000000.

external SDRAM

This is located at address 0xa0000000 of the memory space, ending at location 0xa2000000. For RAM startup, available SRAM starts at location 0xa1100000, with the bottom 1Mbyte reserved for use by RedBoot.

on-chip peripherals

These are accessible via location 0xE0000000 onwards. Descriptions of the contents can be found in the LPC2468 User Manual.

XVI. ST STR7XX variant HAL

Overview

Name

eCos Support for the ST Microelectronics STR7XX ARM microcontrollers —
Overview

Description

The ST STR7XX series of ARM microcontrollers is supported by eCos with an eCos processor variant HAL and a number of device drivers supporting some of the on-chip peripherals. These include device drivers for the on-chip flash, serial and watchdog devices. In addition it provides common functionality and definitions that STR7XX based platform ports may require, as well as definitions useful to application developers.

This documentation covers the STR7XX functionality provided but should be read in conjunction with the specific HAL documentation for the platform port. That documentation will cover issues that are platform-specific and are not covered here, and may also describe differences that override or supersede what the STR7XX variant HAL provides. The areas that are specific to platform HALs and not the STR7XX variant HAL include:

- memory map and related configuration and setup
- memory remapping
- Clock parameters
- GPIO setup
- Any special handling for external interrupts, or additional interrupts
- Diagnostic I/O baud rates
- Additional diagnostic I/O devices, if any
- LED/LCD control

On-chip Subsystems and Peripherals

Name

On-chip Subsystems and Peripherals — Hardware Support

Hardware support

On-chip memory

The ST STR7XX parts include on-chip SRAM, and on-chip FLASH. The RAM consists of a single 64KiB block. The FLASH comprises a block of program memory which is either 64KiB, 128KiB or 256KiB in size depending on model, plus a 16KiB area of higher durability data memory. There is also support in some models for external SRAM and flash, which eCos may use where available.

Typically, an eCos platform HAL port will expect a GDB stub ROM monitor or RedBoot image to be programmed into either the external FLASH or the STR7XX on-chip ROM memory for development, and the board would boot this image from reset. The stub ROM/RedBoot provides GDB stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using serial interfaces or other debug channels. The JTAG interface may also be used for development if a suitable JTAG device is available. If RedBoot is present it may also be used to manage the on-chip and external flash memory. For production purposes, applications are programmed into the external or on-chip ROM and will be self-booting.

Serial I/O

The STR7XX variant HAL supports basic polled HAL diagnostic I/O over any of the on-chip serial devices. There is also a fully interrupt-driven serial device driver suitable for eCos applications for all on-chip serial devices. The serial driver consists of an eCos package: `CYGPKG_IO_SERIAL_ARM_STR7XX` which provides all support for the STR7XX on-chip serial devices. Using the HAL diagnostic I/O support, any of these devices can be used by the ROM monitor or RedBoot for communication with GDB. If a device is needed by the application, either directly or via the serial driver, then it cannot also be used for GDB communication using the HAL I/O support. An alternative serial port should be used instead.

The STR7XX UARTs only provide the minimal TX and RX data lines; hardware flow control using RTS/CTS is not supported. The eCos device drivers have been extended to permit the use of a pair of GPIO lines as flow control lines. It is the responsibility of the platform HAL to enable this functionality and define the GPIO lines to be used in this way.

I²C Support

The I²C[®] driver uses the STR7XX's internal support. This is controlled within the STR7XX variant HAL. The `CYGPKG_HAL_STR7XX_I2C` CDL component controls whether the I²C driver is enabled. Within that component, there are two sub-options:

- `CYGNUM_HAL_STR7XX_I2C_BUS0_CLOCK` sets the speed of the I²C bus 0 clock in Hz. This is usually 100kHz, but can be set up to 400kHz (fast mode) if the devices on the bus support this speed. Other values below 400kHz

can also be chosen, subject to the accuracy of the clock waveform generation parameters.

- `CYGNUM_HAL_STR7XX_I2C_BUS1_CLOCK` sets the speed of the I²C bus 1 clock in Hz. This is usually 100kHz, but can be set up to 400kHz (fast mode) if the devices on the bus support this speed. Other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.

The I²C driver is accessed via the generic I²C driver package `CYGPKG_IO_I2C`. Documentation for its API may be found elsewhere.

This driver only operates in interrupt mode. It does not operate in polled mode, and thus does not operate when interrupts are disabled. It cannot therefore be used in an initialization context, before the eCos kernel thread scheduler starts, and it cannot be used with RedBoot.

Watchdog

A device driver is included for the on-chip watchdog device. This driver allows the use of the standard eCos watchdog API accessible with the `CYGPKG_IO_WATCHDOG` eCos package. If the watchdog is not reset within a time period defined in the watchdog device driver CDL, then the system is automatically reset.

The watchdog device is also used to implement reset functionality, it may also be called directly by applications using the following function:

```
#include <cyg/hal/hal_diag.h>
extern void hal_str7xx_reset_cpu(void);
```

Interrupt controller

eCos manages the on-chip Enhanced Interrupt Controller (EIC). The EIC is configured to use interrupts in non-vector mode, although the vector mechanism is used to aid interrupt source decoding. External interrupts controlled by the XTI unit are also decoded into individual vectors.

Timer 0 is used to implement the eCos system clock. Timer-based profiling support is implemented using timer 1. If the gprof package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then timer 1 is reserved for use by the profiler. Timers 2 and 3 are free for use by applications.

Other

Other on-chip devices (SPI, USB, CAN, HDLC etc.) are not touched by the STR7XX variant HAL and unless used by the platform HAL are free for use for applications.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This section covers any remaining items of note related to the STR7xx variant support, not covered in previous sections.

LEDs

If a platform port has support for the display of values on LEDs, that support is standardised to be accessible from C with the following function:

```
#include <cyg/infra/hal_diag.h>
extern void hal_diag_led(int leds);
```

Clock Control

The platform HAL must provide the input clock frequency (CYGARC_HAL_STR7XX_INPUT_CLOCK) in its CDL file.

STR7XX definitions

The STR7XX variant HAL port includes the header file `var_io.h` which provides useful register definitions used by eCos, that can also be freely used by applications. It includes only limited register definitions for subsystems unused by eCos.

It may be found in the `include/cyg/hal` directory relative to your configuration's install tree, or alternatively in the source repository at `hal/arm/str7xx/var/VERSION/include/var_io.h`. However it should be properly included by applications by using:

```
#include <cyg/hal/hal_io.h>
```

to allow for platform HALs to augment or override any relevant definitions.

Power Control

The kernel idle thread is scheduled to run when the system has no other tasks able to run. The idle thread can call a HAL supplied macro to place the chip into an appropriate power saving mode instead of just going around a busy loop. The STR7XX variant HAL defines the `HAL_IDLE_THREAD_ACTION` macro to use the STR7XX power

control support to place the chip into `WFI` mode which will stop the processor clock, without disabling the on-chip peripherals. This state continues until an interrupt is received.

Further power saving can be achieved by reducing the system clock frequencies using `hal_str7xx_clocks_setup()` described above. This function only changes the clock frequencies; it may also be necessary to change the values of dividers in various peripherals to compensate. There are several routines supplied in the HAL to do this.

Calling `hal_str7xx_uart_reinit()` will cause all the UART baud rate dividers to be reset to match the current value of `PCLK1`. Note that there are limitations to the range of baud rates that can be set, `PCLK1` must be at least 16 times the required rate. Also, the resolution of the baud rate divider may make certain baud rates less accurate at different `PCLK1` frequencies.

Calling `hal_str7xx_watchdog_init(CYGNUM_DEVS_WATCHDOG_ARM_STR7XX_DESIRED_TIMEOUT_US)` will cause the watchdog to be reinitialized with a timeout based on the current value of `PCLK2`. The resolution of the prescaler and the size of the 16 bit counter may render certain watchdog timeouts unachievable at some clock rates.

Calling `HAL_CLOCK_INITIALIZE(CYGNUM_HAL_RTC_PERIOD)` will cause the main system timer to be reinitialized based on the current value of `PCLK2`.

Power Management

Name

Power Management — Details

Synopsis

```
#include <cyg/hal/hal_io.h>

void hal_str7xx_clocks_setup(int index);
void hal_str7xx_set_clock_speed(int index);
cyg_uint32 hal_str7xx_get_clock_speed(void);
cyg_bool hal_str7xx_mode_stop(void);
void hal_str7xx_mode_standby(void);
cyg_uint32 hal_str7xx_startup_mode(void);
void hal_str7xx_uart_setbaud(cyg_uint32 uart, cyg_uint32 baud);
void hal_str7xx_uart_reinit(void);
void hal_str7xx_i2c_init(cyg_uint32 bus, cyg_uint32 clock);
void hal_str7xx_i2c_reinit(void);
void hal_str7xx_watchdog_init(cyg_uint32 timeout);
void hal_str7xx_can_init(cyg_uint32 devno, cyg_uint32 clock);
void hal_str7xx_can_reinit(void);
void hal_str7xx_adc_init(cyg_uint32 rate);
void hal_str7xx_adc_reinit(void);
void hal_str7xx_rtc_init(void);
void hal_str7xx_rtc_alarm_set(cyg_uint32 secs);
void hal_str7xx_rtc_alarm_cancel(void);
cyg_uint32 hal_str7xx_rtc_counter(void);
cyg_uint32 hal_str7xx_rtc_counter_set(cyg_uint32 secs);
```

Description

The STR7XX variant HAL provides support for managing the power consumption of the device. This consists of a collection of functions that may be used to adjust clock frequencies, system modes and other aspects of the device. These routines mainly comprise a "kit of parts" from which applications may construct their own power management policy. The reader is referred to the STR7XX hardware documentation for full details of clock and power management.

The main function is `hal_str7xx_clocks_setup(index)` which controls the frequencies of the main clocks: MCLK, which supplies the CPU and memories; PCLK1, which supplies APB1 including the I2C, SPI and UARTs; and PCLK2, which supplies APB2 including IO ports, Timers, RTC etc. The single argument to this function is an index into a table of `hal_str7xx_clock_params` structures, which is defined by the platform HAL. Each entry in the table has the following structure:

```
typedef struct
{
```

```

char      *name;                // Name string
cyg_uint8 clk2_divider;         // 1, 2
cyg_uint8 pll1_multiplier;     // 12, 16, 20, 24
cyg_uint8 pll1_divider;       // 1..7
cyg_uint8 rclk_select;         // RCLK_SELECT_* below
cyg_uint8 mclk_divider;        // 1, 2, 4, 8
cyg_uint8 pclk1_divider;       // 1, 2, 4, 8
cyg_uint8 pclk2_divider;       // 1, 2, 4, 8
} hal_str7xx_clock_params;

#define RCLK_SELECT_CLK2      0
#define RCLK_SELECT_CLK2_16  1
#define RCLK_SELECT_PLL1     2
#define RCLK_SELECT_AF       3

__externC cyg_uint32 hal_str7xx_clock_param_index;

```

Each entry in the table corresponds to a single configuration of the clock hardware. Some care must be taken in specifying these entries, the resulting clocks will depend on the system input clock and the various multipliers and dividers; many configurations will result in out of range or otherwise illegal clock frequencies. See the manuals for the STR7XX variant and the platform for details.

It is recommended that the *clk2_divider* field is always set to 2. This causes the oscillator input to be divided by 2 and provides a more stable input to the rest of the clock circuitry. Also, some registers in the RCCU are only accessible if MCLK is equal to RCLK, so the *mclk_divider* field should always be 1. These restrictions may be relaxed in special circumstances.

The last entry in this table should be all zeros, to mark the end of the table. A typical table would appear as follows:

```

const hal_str7xx_clock_params hal_str7xx_clock_param_table[] =
{
    // name           clk/  pll1* pll1/ rclk source           mclk/ pclk1/ pclk2/
    { "32KHz"         ,  2,    0,    0, RCLK_SELECT_AF      ,  1,    1,    1 },
    { "500KHz"        ,  2,    0,    0, RCLK_SELECT_CLK2_16,  1,    1,    1 },
    { "8MHz"          ,  2,    0,    0, RCLK_SELECT_CLK2   ,  1,    1,    1 },
    { "24/6/3MHz"     ,  2,   12,    4, RCLK_SELECT_PLL1   ,  1,    4,    8 },
    { "32/32MHz"       ,  2,   16,    4, RCLK_SELECT_PLL1   ,  1,    1,    1 },
    { "40/10/5MHz"     ,  2,   20,    4, RCLK_SELECT_PLL1   ,  1,    4,    8 },
    { "40/40MHz"       ,  2,   20,    4, RCLK_SELECT_PLL1   ,  1,    1,    1 },
    { "48/12/6MHz"     ,  2,   12,    2, RCLK_SELECT_PLL1   ,  1,    4,    8 },
    { "48/24/12MHz"    ,  2,   12,    2, RCLK_SELECT_PLL1   ,  1,    2,    4 },

    { 0               ,  0,    0,    0,                      0,    0,    0 }
};

cyg_uint32 hal_str7xx_clock_param_index = 5;

```

The naming convention used above is that a single frequency implies that all 3 clocks (MCLK, PCLK1 and PCLK2) are set to the same value. Two frequencies mean that MCLK is set to the first and both PCLK1 and PCLK2 are set to the second. Three frequencies show the values for MCLK/PCLK1/PCLK2 in order.

The variable *hal_str7xx_clock_param_index* indicates the table entry of the parameter set that is currently set. This should be initialized by the platform HAL to the index of the default parameter set, which will be used

during initialization. `hal_str7xx_clocks_setup()` also sets a number of other global variables with the clock rates resulting from the parameter set in use:

```
__externC cyg_uint32 hal_str7xx_pclk1;           // PCLK1 frequency in Hz
__externC cyg_uint32 hal_str7xx_pclk2;           // PCLK2 frequency in Hz
__externC cyg_uint32 hal_str7xx_mclk;           // MCLK frequency in Hz
```

Functions to initialize baud rate generators or prescaler dividers for various devices are also present:

`hal_str7xx_uart_setbaud(uart, baud)` sets the baud rate generator of the given UART to the given baud rate, based on the current value of PCLK1. UARTs are numbered 0 to 3, corresponding to the UARTs available on the device and baud rate is given in Hz. `hal_str7xx_uart_reinit()` causes all UARTs baud rate generators to be reinitialized using the last baud rate setting and the current PCLK1 value. It is usually called after changing the system clocks.

`hal_str7xx_i2c_init(bus, clock)` initializes the clock divider of the given I²C bus to the given value based on the current value of PCLK1. The bus numbers are either 0 or 1, and the clock rate is given in Hz. `hal_str7xx_i2c_reinit()` causes all I²C busses clock dividers to be reinitialized using the last clock rate setting and the current PCLK1 value.

`hal_str7xx_watchdog_init(timeout)` initializes the watchdog timeout based on the current PCLK2 setting. The timeout is given in microseconds. Some care is needed in setting this value since the resolution of the prescaler and the width of the 16 bit counter mean that certain timeouts may not be achievable at different PCLK2 frequencies.

`hal_str7xx_can_init(devno, clock)` initializes the clock divider of the given CAN device to the given baud rate based on the current PCLK1 setting. This function sets the entire Bit Timing Register, including the bit segment lengths and the synchronization jump width as well as the clock divider. While this interface is designed to support multiple CAN devices, the current implementation only supports a single CAN bus. The return value from this function indicates whether the requested clock frequency can be supported: zero if it is, -1 if not. `hal_str7xx_can_reinit()` causes the bit timings for all CAN busses to be reinitialized based on the current value of PCLK1.

`hal_str7xx_adc_init(rate)` initializes the prescaler for the ADC device based on the current PCLK2 setting. The rate argument gives the sample rate for each channel in samples per second. All channels share the same sample rate and are sampled on a round-robin basis. Therefore the combined sample rate, and hence maximum interrupt rate, will be four times this frequency. `hal_str7xx_adc_reinit()` causes the rate to be reinitialized based on the current value of PCLK2.

`hal_str7xx_mode_stop()` puts the STR7XX into STOP mode. Aside from entering STOP mode, all this routine does is set the WKUP-INT bit in the XTI CTRL register so that any of the external interrupt lines may be used to restart the system from STOP. However, it does not configure or unmask these lines. Instead, they may be unmasked and configured using the standard interrupt control API (`cyg_interrupt_unmask()`, `cyg_interrupt_configure()` etc.) It is also possible to configure the RTC to wake the STR7XX from STOP mode. The value returned from this function indicates whether STOP mode was entered: `true` if it was, `false` if not. It is usually adequate to just retry in the case of failed entry.

`hal_str7xx_mode_standby()` puts the STR7XX into STANDBY mode. As with entering STOP mode, it is the responsibility of the caller to configure the external interrupt lines and RTC to bring the system out of STANDBY mode. Exit from STANDBY mode causes the STR7XX to reboot, so if this function returns, then an error has occurred during entry to STANDBY. It is usually adequate to just retry in this case.

Note: At the time of writing, it has not been possible, at least on the STR710-EVAL board, to test RTC wakeup from STANDBY mode. It is believed that this is due to a silicon bug in the version of the STR710 present on the board.

`hal_str7xx_startup_mode()` returns the reason for the last restart. It indicates whether the restart was as a result of one of the following events:

- `STARTUP_MODE_RESET`: Standard power-on reset.
- `STARTUP_MODE_WAKEUP`: External WAKEUP event.
- `STARTUP_MODE_LOW_VOLTAGE`: Low voltage detected.
- `STARTUP_MODE_RTC_ALARM`: RTC alarm.
- `STARTUP_MODE_WATCHDOG`: Watchdog expiry.
- `STARTUP_MODE_SOFTWARE`: Software reset.

`hal_str7xx_set_clock_speed(index)` is a wrapper function that reprograms all the clocks and baud rate generators. The argument is the same index into the platform HAL supplied parameter table as given to `hal_str7xx_clocks_setup()`. After calling that function it also calls `hal_str7xx_uart_reinit()`, `hal_str7xx_i2c_reinit()`, `hal_str7xx_can_reinit()`, `HAL_CLOCK_INITIALIZE()` and `hal_str7xx_watchdog_init()`. Unlike `hal_str7xx_clocks_setup()`, this function checks that the supplied index is valid, and returns false if it is not.

`hal_str7xx_get_clock_speed()` returns the parameter table index given to the last call to `hal_str7xx_clocks_setup()` or `hal_str7xx_set_clock_speed()`.

The HAL also contains functions to control the Real Time Clock, RTC. These are mainly oriented towards using the RTC to resume the system from STOP or STANDBY mode. Before making any other calls to the RTC routines, the application must call `hal_str7xx_rtc_init()` to initialize the device. Calling `hal_str7xx_rtc_alarm_set(secs)` sets the alarm to fire after the given number of seconds. After the alarm had fired, or to prevent it firing, call `hal_str7xx_rtc_alarm_cancel()`. Calling `hal_str7xx_rtc_counter()` returns the current value of the RTC counter, which counts seconds. Function `hal_str7xx_rtc_counter_set(secs)` sets the RTC counter to the given value. These last two function may be used by a wallclock driver to provide time and date functionality.

RedBoot Support

The STR7XX HAL installs a number of RedBoot commands to allow testing of the power management support.

speed [-l] [index]

This command reports and sets the clock speed of the STR7XX. Giving the command on its own, with no arguments, lists the available speed settings:

```
RedBoot> speed
0          32KHz
1          500KHz
2          8MHz
```



```

3      24/6/3MHz
4      40/10/5MHz
* 5      48/12/6MHz
6      48/24/12MHz

```

RedBoot>

The index numbers on the left are used as arguments to the **speed** command. The names on the right correspond to the clock parameter set names described before. The parameter set currently in force is indicated by an asterisk in the first column. If the -1 option is given, then more details of the parameter sets are given, together with the current settings of MCLK, PCLK1 and PCLK2:

RedBoot> **speed -1**

MCLK: 48000000, PCLK1 12000000, PCLK2 6000000

C	Ix	Name	CLK/	PLL1*	PLL1/	RCLK	MCLK/	PCLK1/	PCLK2/
	0	32KHz	2	0	0	AF	1	1	1
	1	500KHz	2	0	0	CLK2/16	1	1	1
	2	8MHz	2	0	0	CLK2	1	1	1
	3	24/6/3MHz	2	12	4	PLL1	1	4	8
	4	40/10/5MHz	2	20	4	PLL1	1	4	8
*	5	48/12/6MHz	2	12	2	PLL1	1	4	8
	6	48/24/12MHz	2	24	2	PLL1	2	4	8

RedBoot>

Supplying the **speed** command with the index number of a parameter set will change the STR7XX to use that set of clock parameters:

RedBoot> **speed 3**

Set clock speed 3, please wait...

Now running at new speed

RedBoot> **speed -1**

MCLK: 32000000, PCLK1 32000000, PCLK2 32000000

C	Ix	Name	CLK/	PLL1*	PLL1/	RCLK	MCLK/	PCLK1/	PCLK2/
	0	32KHz	2	0	0	AF	1	1	1
	1	500KHz	2	0	0	CLK2/16	1	1	1
	2	8MHz	2	0	0	CLK2	1	1	1
	3	24/6/3MHz	2	12	4	PLL1	1	4	8
*	4	32/32MHz	2	16	4	PLL1	1	1	1
	5	40/10/5MHz	2	20	4	PLL1	1	4	8
	6	40/40MHz	2	20	4	PLL1	1	1	1
	7	48/12/6MHz	2	12	2	PLL1	1	4	8
	8	48/24/12MHz	2	12	2	PLL1	1	2	4

RedBoot>

Note that setting too low a speed may result in RedBoot not being able to program the serial baud rate generator to maintain the current speed.

stop

The **stop** command puts the STR7XX into STOP mode. The RTC will be programmed to wake the system up after 5 seconds. The device may also be woken up before that timeout using the WAKEUP line, if it is connected to a switch (as it is on the STR710-EVAL board).

standby

The **standby** command puts the STR7XX into STANDBY mode. The RTC will be programmed to wake the system up after 5 seconds. The device may also be woken up before that timeout using the WAKEUP line, if it is connected to a switch (as it is on the STR710-EVAL board).

XVII. STR7XX ADC Driver

STR7XX ADC Driver

Name

STR7XX — ADC Driver

Description

This driver supports the ADC devices available in some variants of the ST STR7XX family of microprocessors.

Sample Size

The STR7XX ADC produces 12 bit samples. Therefore this driver sets `CYGNUM_IO_ADC_SAMPLE_SIZE` to 12. This will cause the generic layer to define `cyg_adc_sample_t` as a 16 bit value.

Sample Rates

The ADC hardware is limited to a maximum of 1K samples per channel. Since channels are sampled on a round-robin basis at 4 times this rate, this means that the total sample rate is 4K samples per second.

The option `CYGNUM_DEVS_ADC_ARM_STR7XX_DEFAULT_RATE` defines a default sample rate and is initially set to 500.

Configuration

For each channel *X* supported the CDL script provides the following configuration options:

`cdl_component CYGPKG_DEVS_ADC_ARM_STR7XX_CHANNELX`

This defines whether the channel is included.

`cdl_option CYGDAT_DEVS_ADC_ARM_STR7XX_CHANNELN_NAME`

This defines the name of the channel.

`cdl_option CYGNUM_DEVS_ADC_ARM_STR7XX_CHANNELX_BUFSIZE`

This defines the size of the channel's sample buffer, in samples.

XVIII. ST STR710-EVAL Board HAL

Overview

Name

eCos Support for the ST STR710-EVAL Board — Overview

Description

The ST STR710-EVAL Board is fitted with an STR710FZ2T6 microcontroller to provide a development environment for all STR71X microcontrollers. The board is fitted with 4MiB of external RAM and 4MiB of external FLASH memory. The board has two 9-pin RS-232 serial interfaces connected to two of the STR710 on-chip UARTs, LEDs, and LCD display, and a JTAG debug interface. Refer to the board documentation and the STR7XX documentation for full details.

For typical eCos development, a RedBoot image is programmed into the external FLASH and the switches set so that the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger using either UART0 or UART1.

This documentation describes platform-specific elements of the STR710-EVAL Board support within eCos. The STR7XX documentation covers various topics including HAL support common to STR7XX variants, and on-chip device support. This document complements the STR7XX documentation.

Supported Hardware

The STR7XX has two on-chip memory regions. A RAM region of 64KiB is present at 0x20000000, which is mapped to 0x00000000 after booting. A FLASH region is present at 0x40000000 and is comprised of 64KiB, 128KiB or 256KiB of program memory plus 16KiB of higher durability data flash. The STR710FZ2T6 on the STR710-EVAL board is equipped with 256KiB.

On-board memory consists of 4MiB of SRAM mapped to 0x62000000 and 4MiB of FLASH mapped to 0x60000000. During booting the external flash is mapped to 0x00000000 but will be replaced with the internal flash for normal execution.

The STR7XX variant HAL includes support for the four on-chip serial devices which are [documented in the variant HAL](#). Only two of these serial devices are connected to external connectors on the board, so only these are normally usable.

The STR710-EVAL board port includes support for the on-chip watchdog and interrupt controller. This support is documented in the [STR7XX variant HAL](#).

Tools

The STR710-EVAL Board port is intended to work with GNU tools configured for an arm-elf target. Thumb mode is supported. The original port was done using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.16.

Setup

Name

Setup — Preparing the STR710-EVAL Board for eCos Development

Overview

In a typical development environment, the STR710-EVAL board boots from external flash into RedBoot. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from external FLASH	redboot_ROM.ecm	redboot_ROM.bin
ROM_INT	RedBoot running from internal FLASH	redboot_ROM_INT.ecm	redboot_ROM_INT.bin
ROM_INT_EXTRAM	RedBoot running from internal FLASH, using external RAM for DATA, BSS and heap	redboot_ROM_INT_EXTRAM.ecm	redboot_ROM_INT_EXTRAM.bin
RAM	RedBoot running from external RAM	redboot_RAM.ecm	redboot_RAM.bin
JTAG	RedBoot running from external RAM, loaded via JTAG	redboot_JTAG.ecm	redboot_JTAG.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. This baud rate can be changed via the configuration option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and rebuilding the stubrom.

Under normal circumstances, RedBoot runs in-place from the external Flash. The RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged. The ROM_INT version does not contain support for the flash filesystem or flash config since it uses only internal RAM, which is not large enough for the necessary data structures. The ROM_INT_EXTRAM version uses external RAM instead of internal RAM and is therefore able to contain a full implementation of the flash filesystem and configuration.

Initial Installation

Two mechanisms are described below to program RedBoot into the external Flash. Both of them require a JTAG

device. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.str710eval.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/str710eval/VERSION/misc` relative to the root of your eCos installation.
5. Place the `bdi2000.str710eval.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
6. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.str710eval.cfg` configuration file at the appropriate point of this process.

Preparing the STR710-EVAL board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.
4. Locate switches SW13, SW14 and SW15 on the board, they are in a bank of 3 next to the processor. Set all these switches to the 2-3 position, this is the position away from the processor. In due course this will ensure that the board boots RedBoot from the external Flash device.
5. Power up the STR710-EVAL board.
6. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
Core#0>
```

7. Confirm correct connection with the BDI2000 with the **reset** command as follows:

```
Core#0> reset
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
```

```

- TARGET: BDI removes TRST
- TARGET: Bypass check 0x00000001 => 0xFFFFFFFF
- TARGET: JTAG exists check failed
- TARGET: Remove RESET and try again
- TARGET: BDI waits for RESET inactive
- TARGET: Bypass check 0x00000001 => 0x00000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x3F0F0F0F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>

```

8. Locate the `redboot_ROM.bin` image within the `loaders` subdirectory of the base of the eCos installation.

9. Copy the `redboot_ROM.bin` file into a location on the host computer accessible to its TFTP server.

Method 1 - Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash. This method uses the built-in capabilities of the BDI2000.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.
2. Cut and paste the following commands into the BDI2000 telnet session. They are used to unlock the relevant Flash blocks that will contain RedBoot.

```

Core#0>unlock 0x60000000 0x2000 8
Unlocking flash at 0x60000000
Unlocking flash at 0x60002000
Unlocking flash at 0x60004000
Unlocking flash at 0x60006000
Unlocking flash at 0x60008000
Unlocking flash at 0x6000a000
Unlocking flash at 0x6000c000
Unlocking flash at 0x6000e000
Unlocking flash passed
Core#0>unlock 0x60010000 0x10000 2
Unlocking flash at 0x60010000
Unlocking flash at 0x60020000
Unlocking flash passed

```

3. Erase the 8 initial 8Kbyte sized Flash blocks, and the following 2 64Kbyte Flash blocks with the following command (the blocks to erase are defined in the `.cfg` file):

```

Core#0>erase
Erasing flash at 0x60000000
Erasing flash at 0x60002000

```

```
Erasing flash at 0x60004000
Erasing flash at 0x60006000
Erasing flash at 0x60008000
Erasing flash at 0x6000a000
Erasing flash at 0x6000c000
Erasing flash at 0x6000e000
Erasing flash at 0x60010000
Erasing flash at 0x60020000
Erasing flash passed
Core#0>
```

4. Program the RedBoot image into Flash with the following command, replacing */RBPATH* with the location of the redboot_ROM.bin file relative to the TFTP server root directory:

```
Core#0>prog 0x60000000 /RBPATH/redboot_ROM.bin bin
Programming /RBPATH/redboot_ROM.bin , please wait ....
Programming flash passed
Core#0>
```

This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port. RedBoot's Flash configuration can be initialized using the [same procedure as required in Method 2 below](#).

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into External Flash with RAM RedBoot

With this approach, the BDI2000 is used to load a RedBoot image into RAM, which can then in turn be used to load and program a ROM RedBoot image into Flash.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROM RedBoot.

Loading a RAM RedBoot

1. Locate the redboot_JTAG.bin image within the loaders subdirectory of the base of the eCos installation.
2. Copy the redboot_JTAG.bin file into a location on the host computer accessible to its TFTP server.
3. With the BDI2000 telnet interface, execute the following command, replacing */RBPATH* with the location of the redboot_JTAG.bin file relative to the TFTP server root directory:

```
Core#0>load 0x62000000 /RBPATH/redboot_JTAG.bin bin
Loading /RBPATH/redboot_JTAG.bin , please wait ....
Loading program file passed
Core#0>
```

4. Run the loaded RAM RedBoot:

```
Core#0>go 0x62000000
Core#0>
```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
+RedBoot(tm) bootstrap and debug environment [JTAG]
Non-certified release, version UNKNOWN - built 17:29:34, Apr 10 2006

Platform: ST STR710-EVAL Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x62000000-0x62400000, [0x62019a20-0x623ed000] available
      0x00000000-0x00010000, [0x00000000-0x00010000] available
FLASH: 0x40000000-0x40c40000, 4 x 0x2000 blocks, 1 x 0x8000 blocks, 3 x 0x10000 blocks, 8 x 0x10000 blocks
FLASH: 0x60000000-0x603fffff, 8 x 0x2000 blocks, 63 x 0x10000 blocks
RedBoot>
```

Note: It is also possible to use the RAM startup version of RedBoot and the `redboot_RAM.bin` file instead of `redboot_JTAG.bin` above. If so, then the address to the **load** command must be `0x62020000`, as must be the address to the **go** command.

RedBoot Flash Configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a JTAG or RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROM RedBoot loaded using [Method 1](#).

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0x603f0000-0x603fffff: .
... Erase from 0x603f0000-0x603fffff: .
... Program from 0x623f0000-0x62400000 to 0x603f0000: .
... Locking from 0x603f0000-0x603fffff: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Console baud rate: 38400
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x603f0000-0x603fffff: .
... Erase from 0x603f0000-0x603fffff: .
... Program from 0x623f0000-0x62400000 to 0x603f0000: .
... Locking from 0x603f0000-0x603fffff: .
RedBoot>
```

Loading and programming the ROM RedBoot

This section describes the steps required to load the ROM RedBoot via the serial line and program it into Flash.

1. Load the RedBoot ROM binary image from the serial line. Use the following command:

```
RedBoot> load -r -m y -b {%freememlo}
CRaw file loaded 0x62038c00-0x6204eacb, assumed entry at 0x62038c00
xyzModem - CRC mode, 704(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

2. Finally install the loaded image into Flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Unlocking from 0x60000000-0x6001ffff: .....
... Erase from 0x60000000-0x6001ffff: .....
... Program from 0x62038c00-0x6204eacc to 0x60000000: .....
... Locking from 0x60000000-0x6001ffff: .....
... Unlocking from 0x603f0000-0x603fffff: .
... Erase from 0x603f0000-0x603fffff: .
... Program from 0x623f0000-0x62400000 to 0x603f0000: .
... Locking from 0x603f0000-0x603fffff: .
RedBoot>
```

It is also possible to use the **fis write** command to write the image into Flash, but if so, the relevant Flash blocks must also be explicitly unlocked with the command:

```
RedBoot> fis unlock -f 0x60000000 -l 0x20000
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port.

```
+RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 10:22:53, Apr 10 2006
```

```
Platform: ST STR710-EVAL Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited
```

```
RAM: 0x62000000-0x62400000, [0x62004910-0x623ed000] available
      0x00000000-0x00010000, [0x00000000-0x00010000] available
FLASH: 0x40000000-0x40c40000, 4 x 0x2000 blocks, 1 x 0x8000 blocks, 3 x 0x10000 blocks, 8 x 0x10000 blocks
FLASH: 0x60000000-0x603fffff, 8 x 0x2000 blocks, 63 x 0x10000 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 3 - Program RedBoot into Internal Flash with RAM RedBoot

This is a variant of Method 2, which puts RedBoot into the internal flash of the STR710, rather than the external M28W320CB flash device.

The first part of this is exactly the same as Method 2: load a RAM version of RedBoot as described in the [Loading a RAM RedBoot](#) section and configure is as described in the [RedBoot Flash Configuration](#) section. Now load the redboot_ROM_INT.bin binary image from the serial line as follows:

```
RedBoot> load -r -m y -b %{freememlo}
CRaw file loaded 0x6201a800-0x6202c733, assumed entry at 0x6201a800
xyzModem - CRC mode, 577(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Install the loaded image into Flash:

```
RedBoot> fis write -f 0x40000000 -b %{freememlo} -l 0x20000
* CAUTION * about to program FLASH
          at 0x40000000..0x4001ffff from 0x6201a800 - continue (y/n)? y
... Erase from 0x40000000-0x4001ffff: .....
... Program from 0x6201a800-0x6203a800 to 0x40000000: .....
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port.

```
+
RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 18:41:12, Jun  1 2006

Platform: ST STR710-EVAL Board (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x00000000-0x00010000, [0x000048b0-0x00010000] available
FLASH: 0x40000000-0x40c40000, 4 x 0x2000 blocks, 1 x 0x8000 blocks, 3 x 0x10000 blocks, 8 x 0x10000 blocks
FLASH: 0x60000000-0x603fffff, 8 x 0x2000 blocks, 63 x 0x10000 blocks
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot are:

```
$ mkdir redboot_str710eval_rom
$ cd redboot_str710eval_rom
$ ecosconfig new str710eval redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/str7xx/str710eval/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

Setup

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - ROM, RAM or JTAG - may be similarly built by choosing the appropriate alternative `.ecm` file.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The STR710-EVAL Board platform HAL package is loaded automatically when eCos is configured for an `str710eval` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The STR710-EVAL Board platform HAL package supports four separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into external Flash at location 0x60000000 and uses external RAM at location 0x62000000. `arm-elf-gdb` is then used to load a RAM startup application into memory from 0x62020000 and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into external ROM at location 0x60000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x20000000 region and map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization.

ROM_INT

This startup type can be used for finished applications which will be programmed into internal Flash at location 0x40000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer control to the 0x40000000 region and map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization.

This startup is enabled by setting the `CYGHWR_HAL_STR7XX_FLASH_INTERNAL` option, when `CYG_HAL_STARTUP` is set to `ROM`.

ROM_INT_EXTRAM

This startup type can be used for finished applications which will be programmed into internal Flash at location 0x40000000. The application will be self-contained with no dependencies on services provided by other software. The program expects to boot from reset with ROM mapped at location zero. It will then transfer

control to the 0x40000000 region and map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization. The application will also use external RAM at 0x62000000 for its DATA, BSS and heap, rather than the internal SRAM.

This startup is enabled by setting the `CYGHWR_HAL_STR7XX_FLASH_INTERNAL` option, when `CYG_HAL_STARTUP` is set to ROM and additionally setting `CYGHWR_HAL_ARM_STR710EVAL_EXT_RAM`.

JTAG

This is the startup type used to build applications that are loaded via a JTAG interface. The application will be self-contained with no dependencies on services provided by other software. The program expects to be loaded from 0x62000000 and entered at that address. It will then map internal RAM to location zero. eCos startup code will perform all necessary hardware initialization and the system will be in a condition suitable for loading and running RAM applications.

The Stubrom and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then serial port 0 will be claimed for HAL diagnostics.

Flash Drivers

The STR710-EVAL board contains a 4Mbyte ST M28W320CB parallel Flash device. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the STR710-EVAL board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

A driver is also present for the internal Flash. The package `CYGPKG_DEVS_FLASH_STR7XX` contains all the code necessary to support this memory and the platform HAL package contains definitions that customize the driver to the STR710-EVAL board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The STR710-EVAL board use the STR7XX's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_STR7XX` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_STR7XX_DESIRED_TIMEOUT_US` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

UART Serial Driver

The STR710-EVAL boards use the STR7XX's internal UART serial support. As well as the polled HAL diagnostic interface, there is also a `CYGPKG_IO_SERIAL_ARM_STR7XX` package which contains all the code necessary to support interrupt-driven operation with greater functionality. All four UARTs can be supported by this driver, although only UARTs 0 and 1 are actually routed to external connectors. Note that it is not recommended to enable this driver on the port used for HAL diagnostic I/O. This driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

The STR7XX UARTs only provide the minimal TX and RX data lines; hardware flow control using RTS/CTS is not supported. The eCos device drivers have been extended to permit the use of a pair of GPIO lines to be used as flow control lines. These must be defined by the platform for each UART using the following CDL options:

`CYGHWR_HAL_ARM_STR7XX_UARTX_RTS`

This value encodes which PIO line will be used as the RTS line for UARTX. The value of this option is an invocation of the macro `UART_PIO()`, which takes three arguments: the first is the PIO port number and the second is the bit number in that port for the PIO line. The third argument gives the polarity of the line, 0 if it is active low, 1 if it is active high.

`CYGHWR_HAL_ARM_STR7XX_UARTX_CTS`

This value encodes which PIO line will be used as the CTS line for UARTX. `UART_PIO()` takes three arguments: the first is the PIO port number and the second is the bit number in that port for the PIO line. The third argument gives the polarity of the line, 0 if it is active low, 1 if it is active high.

`CYGHWR_HAL_ARM_STR7XX_UARTX_CTS_INT`

This must be the name of the interrupt vector, from `var_intr.h`, that corresponds to the PIO bit selected for CTS. It is essential that the PIO bit selected be capable of generating an interrupt, so only those that have an XTI interrupt vector can be used. The polarity of the CTS line will decide whether this interrupt occurs on a rising or falling edge.

I²C Support

Support for the two I²C[®] busses is provided by the variant HAL (`CYGPKG_HAL_ARM_STR7XX`). The STR710-EVAL board carries an ST M24C08 I²C serial EEPROM connected to bus 0.

The M24C08 is an 8Kibit device, 1024 bytes. This memory is addressed by using a single byte for the least significant 8 bits of the address plus 2 bits from the device's I²C address. Within eCos, this EEPROM is presented as four separate I²C devices at addresses 0xA8, 0xAA, 0xAC, 0xAE. These are instantiated as four I²C device objects, named `cyg_i2c_str710eval_m24c08_0`, `cyg_i2c_str710eval_m24c08_1`, `cyg_i2c_str710eval_m24c08_2` and `cyg_i2c_str710eval_m24c08_3` respectively.

A test application for use with the EEPROM is provided within the `tests` subdirectory of the `CYGPKG_HAL_ARM_STR7XX_STR710EVAL` package. This test communicates with the I²C EEPROM on the board to perform read and write operations using I²C. Since it overwrites the contents of the EEPROM, this test is not built by default. It may be built by enabling the configuration option `CYGBLD_HAL_ARM_STR7XX_STR710EVAL_TEST_M24C08`.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. The description of the clock-related options may be found in the STR7XX variant HAL documentation.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos.

The option `"-mcpu=arm7tdmi"` should be set for all compilations for this platform.

There are two flags that are used if Thumb mode is to be supported:

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the STR7XX only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.str710eval.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs.

The `bdi2000.str710eval.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the `bdiGDB` interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.str710eval.cfg` file, and halt the target. This behaviour is repeated with the **reset** command.

If the board is reset when in '**reset**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) JTAG applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
Core#0>load 0x62000000 test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
```

JTAG debugging support

```
Core#0>go 0x620000000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of JTAG applications

JTAG applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB (\$O) packets. Both of these settings are made automatically if the JTAG startup type is selected.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the STR710-EVAL Board hardware, and should be read in conjunction with that specification. The STR710-EVAL Board platform HAL package complements the ARM architectural HAL and the STR7XX variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize many of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM and JTAG startup, the HAL will perform additional initialization, programming the various internal registers including the PLL, peripheral clocks, GPIO pins and memory mapping control to map internal RAM 0x0. The details of the early hardware startup may be found in the header `cyg/hal/hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

External RAM

This is located at address 0x62000000 of the memory space, and is 4MiB long. For ROM applications, all of RAM is available for use. For RAM startup applications, RAM below 0x62020000 is reserved for RedBoot and the remainder is available for the application.

External ROM

This is located at address 0x60000000 of the memory space. If switches SW13, SW14 and SW15 are all set to 2-3 this region will be mapped to 0x00000000 at reset. This region is 4MiB in size. RedBoot is normally programmed into this memory and the rest managed by the FIS flash file system.

Internal RAM

This is located at address 0x20000000 of the memory space, and is 64KiB in size. Normally this RAM area will be mapped to location 0x00000000 after bootstrap. The CPU vector table and the eCos VSR table occupy the bottom 64 bytes. The virtual vector table starts at 0x00000050 and extends to 0x00000150. The remainder of internal RAM is available for use by applications.

Internal ROM

This is located at address 0x40000000 of the memory space. If switches SW13 and SW14 are set to 1-2 and SW15 to 2-3 this region will be mapped to 0x00000000 at reset. This region is 256KiB in size. Applications may be configured to run from this memory by setting the `CYGHWR_HAL_STR7XX_FLASH_INTERNAL` option. This memory is not managed by RedBoot's FIS system, but it can be written using the **fis write** command and erased using the **fis erase** command.

on-chip peripherals

These are accessible at locations 0xC0000000 and 0xE0000000 upwards, depending on which APB bus they are on. Descriptions of the contents can be found in the STR7XX User Manual.

XIX. Atmel AT91 Processor Variant Support

Overview

Name

Support for the Atmel AT91 Processor Variant — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91 processor family which includes the AT91R4xxx series, the M42xxx and M55xxx series and the AT91SAM7S, -X and -A series. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, AT91SAM7 variant HAL (where appropriate) and the platform HAL. It provides functionality common to all AT91-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/at91/var` within the eCos source repository.

The AT91 processor HAL package is loaded automatically when eCos is configured for an AT91-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel AT91 processor within this processor HAL package include:

- [AT91-specific hardware definitions](#)
- [Interrupt controller](#)
- [Timer counters](#)
- [Serial UARTs](#)

Support for the interrupt-driven serial, SPI, watchdog and wallclock (RTC) features of the AT91 are also present and can be found in separate packages, outside of this processor HAL.

The watchdog hardware may also be used within this HAL to perform software reset.

Hardware definitions

Name

AT91 hardware definitions — Details on obtaining hardware definitions for AT91

Register definitions

The file `<cyg/hal/var_io.h>` provides definitions related to AT91 subsystems. This file should not be included explicitly, but is included automatically whenever `<cyg/hal/hal_io.h>` is included. This file includes register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, CAN, TWI (I²C[®]), Ethernet, timer counter, RTC, and SPI subsystems, depending on the exact model.

Interrupt Controller

Name

AT91 interrupt controller — Advanced Interrupt Controller definitions and usage

Interrupt Controller Support

The AT91 variant HAL contains generic support for the AIC (GIC on SAM7XA1 and -A2). It queries the interrupt controller to identify the current interrupt and vectors to the matching service routine. If the device supports the SYSTEM interrupt then the devices that raise this interrupt will be queried individually and vectored to their own interrupt handlers. The mapping between interrupts and vector numbers is defined in the `hal_platform_ints.h` file in either the platform HAL or the AT91SAM7 variant HAL.

As indicated above, further decoding is performed on the SYSTEM interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the SYSTEM interrupt will not work as expected. Conversely, masking a decoded derivative of the SYSTEM interrupt will not work as this would mask other SYSTEM interrupts, but masking the SYSTEM interrupt itself will work. On the other hand, unmasking a decoded SYSTEM interrupt *will* unmask the SYSTEM interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

Interrupt Controller Functions

The source file `src/at91_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_IRQ_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function, where the `level` and `up` arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded, and as a result, the `hal_interrupt_acknowledge` function is empty.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Interrupt handling within standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory layouts are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers defined in the platform or AT91SAM7 HAL. Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the interrupt vector number. The `data` argument is extracted from a corresponding table named `hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Timers

Name

Timers — Use of on-chip Timer

Hardware Timer

The eCos kernel system clock is implemented using an on-chip Timer. Depending on the device this will either be a Timer Counter, a Simple Timer, or the Periodic Interval Timer. The kind of device used is determined by the platform HAL. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

The selected Timer is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using timer counter 1 (TC1). If the `gprof` package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then TC1 is reserved for use by the profiler.

Serial UARTs

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

Support is included in this processor HAL package for the AT91's on-chip debug unit UART and serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 38400,8,N,1 with no flow control.

HAL diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the AT91 processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

This driver is contained in the `CYGPKG_IO_SERIAL_ARM_AT91` package. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

The USARTs are named `"/dev/ser0"`, `"/dev/ser1"` and so on. The DEBUG serial port is given the name `"/dev/dbg"`. These names are all configurable.

Serial UARTs

Note that unlike the USART devices, the serial debug port does not support modem control signals such as those used for hardware flow control. In addition, USART devices for a particular platform may also not have these control signals brought out to the physical serial port.

XX. Atmel AT91SAM7 Processor Variant Support

Overview

Name

Support for the Atmel AT91SAM7 Processor Variant — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91SAM7 processor family which includes the AT91SAM7S, -X and -A series. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This HAL package complements the ARM architectural HAL, AT91 variant HAL (where appropriate) and the platform HAL. It provides functionality common to all AT91SAM7-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/at91/at91sam7` within the eCos source repository.

The AT91SAM7 HAL package is loaded automatically when eCos is configured for an AT91SAM7-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel AT91SAM7 within this processor HAL package include:

- [AT91SAM7-specific hardware definitions](#)
- [Interrupt Vector Definitions](#)

Support for the interrupt-driven serial, SPI, watchdog and wallclock (RTC) features of the AT91SAM7 are also present and can be found in separate packages, outside of this processor HAL.

Hardware definitions

Name

AT91SAM7 hardware definitions — Details on obtaining hardware definitions for AT91

Device Definitions

The file `<cyg/hal/plf_io.h>` provides definitions related to AT91SAM7 subsystems. This file should not be included explicitly, but is included automatically whenever `<cyg/hal/hal_io.h>` is included. This file mainly includes base address definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, CAN, TWI (I²C®), Ethernet, timer counter, RTC, and SPI subsystems, depending on the exact model.

Interrupt Vector Definitions

Name

AT91SAM7 interrupt vector definitions — Advanced Interrupt Controller vector definitions

Interrupt Vector Definitions

The file <cyg/hal/hal_platform_ints.h> (located at hal/arm/arm9/at91sam7/VERSION/include/hal_platform_ints.h in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs. The exact set of vectors supported depends on the AT91SAM7 model:

For the AT91SAM7S family:

```
#define CYGNUM_HAL_INTERRUPT_FIQ 0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYS 1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA 2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_ADC 4 // Analog-to-Digital Converter
#define CYGNUM_HAL_INTERRUPT_SPI 5 // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_USART0 6 // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1 7 // USART 1
#define CYGNUM_HAL_INTERRUPT_SSC 8 // Serial Synchronous Controller
#define CYGNUM_HAL_INTERRUPT_TWI 9 // Two-Wire Interface (I2C)
#define CYGNUM_HAL_INTERRUPT_PPMC 10 // PWM Controller
#define CYGNUM_HAL_INTERRUPT_UDP 11 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TC0 12 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1 13 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2 14 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_IRQ0 30 // External IRQ0
#define CYGNUM_HAL_INTERRUPT_IRQ1 31 // External IRQ0

// Interrupts which are multiplexed on to the System Interrupt
#define CYGNUM_HAL_INTERRUPT_PITC 32 // Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_RTTC 33 // Real-Time Timer
#define CYGNUM_HAL_INTERRUPT_PMC 34 // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_MC 35 // Memory Controller
#define CYGNUM_HAL_INTERRUPT_WDTC 36 // Watchdog
#define CYGNUM_HAL_INTERRUPT_RSTC 37 // Reset Controller
#define CYGNUM_HAL_INTERRUPT_DEBUG 38 // Debug Serial Port
```

For the AT91SAM7X family:

```
#define CYGNUM_HAL_INTERRUPT_FIQ 0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYS 1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA 2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB 3 // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_SPI 4 // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_SPI1 5 // Serial Peripheral Interface 1
#define CYGNUM_HAL_INTERRUPT_USART0 6 // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1 7 // USART 1
#define CYGNUM_HAL_INTERRUPT_SSC 8 // Serial Synchronous Controller
```

Interrupt Vector Definitions

```
#define CYGNUM_HAL_INTERRUPT_TWI 9 // Two-Wire Interface (I2C)
#define CYGNUM_HAL_INTERRUPT_PWMC 10 // PWM Controller
#define CYGNUM_HAL_INTERRUPT_UDP 11 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TC0 12 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1 13 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2 14 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_CAN 15 // CAN Controller
#define CYGNUM_HAL_INTERRUPT_EMAC 16 // Ethernet MAC
#define CYGNUM_HAL_INTERRUPT_ADC 17 // Analog-to-Digital Converter
#define CYGNUM_HAL_INTERRUPT_IRQ0 30 // External IRQ0
#define CYGNUM_HAL_INTERRUPT_IRQ1 31 // External IRQ0

// Interrupts which are multiplexed on to the System Interrupt
#define CYGNUM_HAL_INTERRUPT_PITC 32 // Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_RTTC 33 // Real-Time Timer
#define CYGNUM_HAL_INTERRUPT_PMC 34 // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_MC 35 // Memory Controller
#define CYGNUM_HAL_INTERRUPT_WDTC 36 // Watchdog
#define CYGNUM_HAL_INTERRUPT_RSTC 37 // Reset Controller
#define CYGNUM_HAL_INTERRUPT_DEBUG 38 // Debug Serial Port
```

For the AT91SAM7A3:

```
#define CYGNUM_HAL_INTERRUPT_FIQ 0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYS 1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA 2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB 3 // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_CAN0 4 // CAN Controller 0
#define CYGNUM_HAL_INTERRUPT_CAN1 5 // CAN Controller 1
#define CYGNUM_HAL_INTERRUPT_USART0 6 // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1 7 // USART 1
#define CYGNUM_HAL_INTERRUPT_USART2 8 // USART 2
#define CYGNUM_HAL_INTERRUPT_MCI 9 // Multimedia Card Interface
#define CYGNUM_HAL_INTERRUPT_TWI 10 // Two-Wire Interface (I2C)
#define CYGNUM_HAL_INTERRUPT_SPI 11 // Serial Parallel Interface 0
#define CYGNUM_HAL_INTERRUPT_SPI1 12 // Serial Parallel Interface 1
#define CYGNUM_HAL_INTERRUPT_SSC0 13 // Serial Synchronous Controller 0
#define CYGNUM_HAL_INTERRUPT_SSC1 14 // Serial Synchronous Controller 1
#define CYGNUM_HAL_INTERRUPT_TC0 15 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1 16 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2 17 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_TC3 18 // Timer Counter 3
#define CYGNUM_HAL_INTERRUPT_TC4 19 // Timer Counter 4
#define CYGNUM_HAL_INTERRUPT_TC5 20 // Timer Counter 5
#define CYGNUM_HAL_INTERRUPT_TC6 21 // Timer Counter 6
#define CYGNUM_HAL_INTERRUPT_TC7 22 // Timer Counter 7
#define CYGNUM_HAL_INTERRUPT_TC8 23 // Timer Counter 8
#define CYGNUM_HAL_INTERRUPT_ADC0 24 // Analog-to-Digital Converter 0
#define CYGNUM_HAL_INTERRUPT_ADC1 25 // Analog-to-Digital Converter 1
#define CYGNUM_HAL_INTERRUPT_PWMC 26 // PWM Controller
#define CYGNUM_HAL_INTERRUPT_UDP 27 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_IRQ0 28 // External Interrupt 0
#define CYGNUM_HAL_INTERRUPT_IRQ1 29 // External Interrupt 1
```

```
#define CYGNUM_HAL_INTERRUPT_IRQ2 30      // External Interrupt 2
#define CYGNUM_HAL_INTERRUPT_IRQ3 31      // External Interrupt 3

// Interrupts which are multiplexed on to the System Interrupt
#define CYGNUM_HAL_INTERRUPT_PITC      32      // Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_RTTC      33      // Real-Time Timer
#define CYGNUM_HAL_INTERRUPT_PMC      34      // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_MC        35      // Memory Controller
#define CYGNUM_HAL_INTERRUPT_WDTC      36      // Watchdog
#define CYGNUM_HAL_INTERRUPT_RSTC      37      // Reset Controller
#define CYGNUM_HAL_INTERRUPT_DEBUG    38      // Debug Serial Port
```

For the AT91SAM7A1 and AT91SAM7A2:

```
#define CYGNUM_HAL_INTERRUPT_FIQ      0      // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SWIIRQ0  1      // Software Interrupt 0
#define CYGNUM_HAL_INTERRUPT_WD        2      // Watchdog
#define CYGNUM_HAL_INTERRUPT_WT        3      // Watch Timer
#define CYGNUM_HAL_INTERRUPT_USART0    4      // USART 0
#define CYGNUM_HAL_INTERRUPT_USART1    5      // USART 1
#define CYGNUM_HAL_INTERRUPT_CAN3      6      // CAN Controller 3
#define CYGNUM_HAL_INTERRUPT_SPI       7      // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_CAN1      8      // CAN Controller 1
#define CYGNUM_HAL_INTERRUPT_CAN2      9      // CAN Controller 2
#define CYGNUM_HAL_INTERRUPT_ADC0      10     // Analog-to-Digital Converter 0
#define CYGNUM_HAL_INTERRUPT_ADC1      11     // Analog-to-Digital Converter 1
#define CYGNUM_HAL_INTERRUPT_GPT0CH0   12     // General Purpose Timer 0 Channel 0
#define CYGNUM_HAL_INTERRUPT_GPT0CH1   13     // General Purpose Timer 0 Channel 1
#define CYGNUM_HAL_INTERRUPT_GPT0CH2   14     // General Purpose Timer 0 Channel 2
#define CYGNUM_HAL_INTERRUPT_SWIIRQ1   15     // Software Interrupt 1
#define CYGNUM_HAL_INTERRUPT_SWIIRQ2   16     // Software Interrupt 2
#define CYGNUM_HAL_INTERRUPT_SWIIRQ3   17     // Software Interrupt 3
#define CYGNUM_HAL_INTERRUPT_GPT1CH0   18     // General Purpose Timer 1 Channel 0
#define CYGNUM_HAL_INTERRUPT_PWM       19     // PWM Controller
#define CYGNUM_HAL_INTERRUPT_CAN0      20     // CAN Controller 0
#define CYGNUM_HAL_INTERRUPT_UPIO      21     // Unified Parallel IO Controller
#define CYGNUM_HAL_INTERRUPT_CAPT0     22     // Capture 0
#define CYGNUM_HAL_INTERRUPT_CAPT1     23     // Capture 1
#define CYGNUM_HAL_INTERRUPT_ST0       24     // Simple Timer 0
#define CYGNUM_HAL_INTERRUPT_ST1       25     // Simple Timer 1
#define CYGNUM_HAL_INTERRUPT_SWIIRQ4   26     // Software Interrupt 4
#define CYGNUM_HAL_INTERRUPT_SWIIRQ5   27     // Software Interrupt 5
#define CYGNUM_HAL_INTERRUPT_IRQ0      28     // External Interrupt 0
#define CYGNUM_HAL_INTERRUPT_IRQ1      29     // External Interrupt 1
#define CYGNUM_HAL_INTERRUPT_SWIIRQ6   30     // Software Interrupt 6
#define CYGNUM_HAL_INTERRUPT_SWIIRQ7   31     // Software Interrupt 7
```


XXI. Atmel AT91SAM7A2-EK Board Support

Overview

Name

eCos Support for the Atmel AT91SAM7A2-EK — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM7A2-EK Evaluation Kit. The AT91SAM7A2-EK Evaluation Kit contains the AT91SAM7A2 processor, 512KiB of SRAM, 2MiB of parallel NOR Flash memory, external connections for a single serial channel, CAN and LIN ports. eCos support for the devices and peripherals on the AT91SAM7A2 is described below.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. Applications may also be loaded into RAM via a JTAG debugger, or may be programmed into flash.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The parallel NOR flash memory consists of 31 blocks of 16KiBytes each, followed by 8 blocks of 8KiBytes each. In a typical setup, the first 128 KiBytes are reserved for the use of the RedBoot image. The topmost 8 blocks are used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 30 blocks available between 0x40020000 and 0x401EFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port can be used for communication. If RedBoot is installed, it uses the Debug Unit serial device. The serial driver package is loaded automatically when configuring for the AT91SAM7A2-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91`. This driver is also loaded automatically when configuring for the AT91SAM7A2-EK target.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7A2-EK support is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM7A2-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM7A2-EK boards boot from the parallel NOR Flash and run the RedBoot ROM monitor directly. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.bin
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.bin
JTAG	RedBoot running from RAM, loaded via JTAG	redboot_JTAG.ecm	redboot_JTAG.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud.

The ROM version is programmed into flash to boot the system; however, it is not able to reprogram the RedBoot image in flash. The RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. The ELF format image of this JTAG version of RedBoot can be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged.

Initial Installation

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. Two mechanisms are described below to program RedBoot into Flash. Both of them require a JTAG device. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.

2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.at91sam7a2ek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7a2ek/VERSION/misc` relative to the root of your eCos installation.
5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the `bdi2000.at91sam7a2ek.cfg` in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.
8. Open `bdi2000.at91sam7a2ek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the [REGS] section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91sam7a2ek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7A2-EK board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.
4. Power up the AT91SAM7A2-EK board.
5. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
SAM7A2>
```

6. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
SAM7A2> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x1F0F0F0F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
```

```
- TARGET: processing target startup passed
SAM7A2>
```

7. Locate the `redboot_ROM.bin` image within the `loaders` subdirectory of the base of the eCos installation.
8. Copy the `redboot_ROM.bin` file into a location on the host computer accessible to its TFTP server.

Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash. This method uses the built-in capabilities of the BDI2000.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.
2. Erase the 8 initial 8Kbyte sized Flash blocks, and the following 64Kbyte Flash block with the following command:

```
SAM7A>erase
Erasing flash at 0x40000000
Erasing flash at 0x40002000
Erasing flash at 0x40004000
Erasing flash at 0x40006000
Erasing flash at 0x40008000
Erasing flash at 0x4000a000
Erasing flash at 0x4000c000
Erasing flash at 0x4000e000
Erasing flash at 0x40010000
Erasing flash passed
SAM7A>
```

3. Program the RedBoot image into Flash with the following command, replacing `/RBPATH` with the location of the `redboot_ROM.bin` file relative to the TFTP server root directory:

```
SAM7A>prog 0x40000000 /RBPATH/redboot_ROM.bin bin
Programming redboot_ROM.bin , please wait ....
Programming flash passed
SAM7A>
```

This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port. RedBoot's Flash configuration can be initialized using the [same procedure as required in Method 2 below](#).

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into Flash with RAM RedBoot

With this approach, the BDI2000 is used to load a RAM RedBoot image, which can then in turn be used to load and program a ROMRAM RedBoot image into Flash.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot.

Loading a RAM RedBoot

1. Locate the `redboot_JTAG.bin` image within the `loaders` subdirectory of the base of the eCos installation.
2. Copy the `redboot_JTAG.bin` file into a location on the host computer accessible to its TFTP server.
3. With the BDI2000 telnet interface, execute the following command, replacing `/RBPATH` with the location of the `redboot_JTAG.bin` file relative to the TFTP server root directory:

```
SAM7A>load 0x48000000 /RBPATH/redboot_JTAG.bin bin
Loading /RBPATH/redboot_JTAG.bin , please wait ....
Loading program file passed
SAM7A>
```

4. Run the loaded RAM RedBoot:

```
SAM7A>go 0x48000040
SAM7A>
```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database

RedBoot(tm) bootstrap and debug environment [JTAG]
Non-certified release, version UNKNOWN - built 11:03:18, Oct 27 2006

Platform: Atmel AT91SAM7A2-EK (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x48000000-0x48040000, [0x48017078-0x4802d000] available
FLASH: 0x40000000-0x401fffff, 31 x 0x10000 blocks, 8 x 0x2000 blocks
RedBoot>
```

Note: It is also possible to use the RAM startup version of RedBoot and the `redboot_RAM.bin` file instead of `redboot_JTAG.bin` above. If so, then the address to the **load** command must be `0x48008000`, and the start address given to the **go** command should be `0x48008040`.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROM RedBoot loaded using [Method 1](#).

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x40020000-0x401effff: .....
... Erase from 0x401f0000-0x401fffff: .....
... Program from 0x48030000-0x48040000 to 0x401f0000: .....
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
... Erase from 0x401f0000-0x401fffff: .....
... Program from 0x48030000-0x48040000 to 0x401f0000: .....
RedBoot>
```

Loading and programming the ROM RedBoot

This section describes the steps required to load the ROM RedBoot from the TFTP server and program it into Flash.

1. Load the RedBoot ROM binary image using Y-Modem protocol over the serial line. First give this command to RedBoot:

```
RedBoot> load -r -m y -b %{freememlo}
C
RedBoot>
```

Use the Y-Modem protocol support of your terminal emulator to send the `redboot_ROM.bin` file at this point. When the transfer is finished RedBoot will report:

```
Raw file loaded 0x48017400-0x4802abd3, assumed entry at 0x48017400
xyzModem - CRC mode, 627(SOH)/0(STX)/0(CAN) packets, 8 retries
RedBoot>
```

2. Finally install the loaded image into Flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x40000000-0x4001ffff: .....
... Program from 0x48017400-0x4802abd4 to 0x40000000: .....
```

Setup

```
... Erase from 0x401f0000-0x401fffff: .  
... Program from 0x48030000-0x48040000 to 0x401f0000: .  
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port.

```
+  
RedBoot(tm) bootstrap and debug environment [ROM]  
Non-certified release, version UNKNOWN - built 11:41:25, Oct 27 2006  
  
Platform: Atmel AT91SAM7A2-EK (ARM7TDMI)  
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.  
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited  
  
RAM: 0x48000000-0x48040000, [0x48005150-0x4802d000] available  
FLASH: 0x40000000-0x401fffff, 8 x 0x2000 blocks, 31 x 0x10000 blocks  
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7A2-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7a2ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x40000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG

This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7A2-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains the code necessary to support this part and the platform HAL package contains definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7A2-EK board use the AT91SAM7A2's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

USART Serial Driver

The AT91SAM7A2-EK board use the AT91SAM7A2's internal USART serial support as described in the AT91 processor HAL documentation. One serial ports is available: USART 0 which is mapped to virtual vector channel 0 and `"/dev/ser0"`. USART 0 does not support modem control signals such as those used for hardware flow control.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7A2.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7A2 only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91sam7a2ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi2000.at91sam7a2ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the `bdiGDB` interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91sam7a2ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in **'reset halt'** mode (either with the **'reset halt'** or **'reset'** commands, or by pressing the reset button) and the **'go'** command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type **'go'** every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7A2>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7A2>go 0x00201000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi2000.at91sam7a2ek.cfg` configuration on the BDI2000, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB (\$O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial port.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7A2-EK hardware, and should be read in conjunction with that specification. The AT91SAM7A2-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x40000000 of the physical memory space.

RAM

This is located at address 0x48000000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x48008000, with the bottom 32KiB reserved for use by RedBoot.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    416 size  3920
Startup : Interrupt stack used    148 size  4096
Startup : Idlethread stack used     88 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 61.00 microseconds (57 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            10
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
46.93	43.73	50.13	1.71	40%	30%	Create thread
8.96	8.53	9.60	0.51	60%	60%	Yield thread [all suspended]
9.28	8.53	9.60	0.45	70%	30%	Suspend [suspended] thread
10.03	9.60	10.67	0.51	60%	60%	Resume thread
13.87	13.87	13.87	0.00	100%	100%	Set priority
1.49	1.07	2.13	0.51	60%	60%	Get priority
32.00	32.00	32.00	0.00	100%	100%	Kill [suspended] thread
9.07	8.53	9.60	0.53	100%	50%	Yield [no other] thread
16.96	16.00	17.07	0.19	90%	10%	Resume [suspended low prio] thread
9.60	9.60	9.60	0.00	100%	100%	Resume [runnable low prio] thread
12.80	12.80	12.80	0.00	100%	100%	Suspend [runnable] thread
8.96	8.53	9.60	0.51	60%	60%	Yield [only low prio] thread
9.17	8.53	9.60	0.51	60%	40%	Suspend [runnable->not runnable]
32.00	32.00	32.00	0.00	100%	100%	Kill [runnable] thread
23.15	22.40	23.47	0.45	70%	30%	Destroy [dead] thread
46.51	45.87	46.93	0.51	60%	40%	Destroy [runnable] thread
62.29	60.80	69.33	1.41	60%	90%	Resume [high priority] thread
24.07	23.47	32.00	0.59	50%	49%	Thread switch
1.60	1.07	2.13	0.53	100%	50%	Scheduler lock
6.93	6.40	7.47	0.53	100%	50%	Scheduler unlock [0 threads]
6.93	6.40	7.47	0.53	100%	50%	Scheduler unlock [1 suspended]

6.93	6.40	7.47	0.53	100%	50%	Scheduler unlock [many suspended]
6.93	6.40	7.47	0.53	100%	50%	Scheduler unlock [many low prio]
2.67	2.13	3.20	0.53	100%	50%	Init mutex
10.40	9.60	10.67	0.40	75%	25%	Lock [unlocked] mutex
11.73	11.73	11.73	0.00	100%	100%	Unlock [locked] mutex
9.87	9.60	10.67	0.40	75%	75%	Trylock [unlocked] mutex
8.47	7.47	8.53	0.12	93%	6%	Trylock [locked] mutex
1.07	1.07	1.07	0.00	100%	100%	Destroy mutex
62.57	61.87	62.93	0.48	65%	34%	Unlock/Lock mutex
3.53	3.20	4.27	0.46	68%	68%	Create mbox
1.00	0.00	1.07	0.12	93%	6%	Peek [empty] mbox
11.73	11.73	11.73	0.00	100%	100%	Put [first] mbox
1.00	0.00	1.07	0.12	93%	6%	Peek [1 msg] mbox
11.20	10.67	11.73	0.53	100%	50%	Put [second] mbox
1.00	0.00	1.07	0.12	93%	6%	Peek [2 msgs] mbox
11.67	10.67	11.73	0.12	93%	6%	Get [first] mbox
11.60	10.67	11.73	0.23	87%	12%	Get [second] mbox
9.53	8.53	9.60	0.12	93%	6%	Tryput [first] mbox
8.93	8.53	9.60	0.50	62%	62%	Peek item [non-empty] mbox
10.40	9.60	10.67	0.40	75%	25%	Tryget [non-empty] mbox
8.73	8.53	9.60	0.32	81%	81%	Peek item [empty] mbox
9.13	8.53	9.60	0.52	56%	43%	Tryget [empty] mbox
1.13	1.07	2.13	0.12	93%	93%	Waiting to get mbox
1.13	1.07	2.13	0.12	93%	93%	Waiting to put mbox
3.47	3.20	4.27	0.40	75%	75%	Delete mbox
43.40	42.67	43.73	0.46	68%	31%	Put/Get mbox
2.27	2.13	3.20	0.23	87%	87%	Init semaphore
8.60	8.53	9.60	0.12	93%	93%	Post [0] semaphore
9.60	9.60	9.60	0.00	100%	100%	Wait [1] semaphore
8.00	7.47	8.53	0.53	100%	50%	Trywait [0] semaphore
8.33	7.47	8.53	0.33	81%	18%	Trywait [1] semaphore
2.47	2.13	3.20	0.46	68%	68%	Peek semaphore
1.20	1.07	2.13	0.23	87%	87%	Destroy semaphore
39.17	38.40	39.47	0.43	71%	28%	Post/Wait semaphore
3.73	3.20	4.27	0.53	100%	50%	Create counter
1.33	1.07	2.13	0.40	75%	75%	Get counter value
1.33	1.07	2.13	0.40	75%	75%	Set counter value
10.20	9.60	10.67	0.52	56%	43%	Tick counter
1.27	1.07	2.13	0.32	81%	81%	Delete counter
2.27	2.13	3.20	0.23	87%	87%	Init flag
9.13	8.53	9.60	0.52	56%	43%	Destroy flag
7.93	7.47	8.53	0.52	56%	56%	Mask bits in flag
9.53	8.53	9.60	0.12	93%	6%	Set bits in flag [no waiters]
13.87	13.87	13.87	0.00	100%	100%	Wait for flag [AND]
13.60	12.80	13.87	0.40	75%	25%	Wait for flag [OR]
13.93	13.87	14.93	0.12	93%	93%	Wait for flag [AND/CLR]
13.73	12.80	13.87	0.23	87%	12%	Wait for flag [OR/CLR]
0.93	0.00	1.07	0.23	87%	12%	Peek on flag
5.97	5.33	6.40	0.51	59%	40%	Create alarm

```

17.60  17.07  18.13  0.53  100%  50% Initialize alarm
 8.23   7.47   8.53  0.43   71%  28% Disable alarm
16.23  16.00  17.07  0.36   78%  78% Enable alarm
 9.87   9.60  10.67  0.40   75%  75% Delete alarm
11.87  11.73  12.80  0.23   87%  87% Tick counter [1 alarm]
74.13  73.60  74.67  0.53  100%  50% Tick counter [many alarms]
22.67  22.40  23.47  0.40   75%  75% Tick & fire counter [1 alarm]
452.10 450.13 504.53  3.28   96%  96% Tick & fire counters [>1 together]
85.70  85.33  86.40  0.48   65%  65% Tick & fire counters [>1 separately]
58.67  58.67  58.67  0.00  100% 100% Alarm latency [0 threads]
65.02  58.67  72.53  3.94   42%  30% Alarm latency [2 threads]
64.89  58.67  72.53  3.99   42%  35% Alarm latency [many threads]
100.33 100.27 108.80  0.13   99%  99% Alarm -> thread resume latency

10.68  10.67  11.73  0.00                      Clock/interrupt latency

24.98  21.33 146.13  0.00                      Clock DSR latency

274    232    292 (main stack: 804) Thread stack used (1360 total)
      All done, main stack : stack used 804 size 3920
      All done : Interrupt stack used 208 size 4096
      All done : Idlethread stack used 252 size 2048

```

Timing complete - 30940 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM7A2-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXII. Atmel AT91SAM7A3-EK Board Support

Overview

Name

eCos Support for the Atmel AT91SAM7A3-EK — Overview

Description

This document covers the configuration and usage of eCos and GDB Stubs on the Atmel AT91SAM7A3-EK Evaluation Kit. The AT91SAM7A3-EK Evaluation Kit contains the AT91SAM7A3 processor, external connections for one serial channel, USB host/device, MMC card. eCos support for the devices and peripherals on the AT91SAM7A3 is described below.

Application development on this board can take one of several approaches. Applications may be loaded into RAM via a JTAG device; however, the application size is limited by the amount of on-chip RAM: 32KiB. Applications may also be loaded into the on-chip flash memory where the RAM limit will only apply to the data portion of the application. Finally, it is possible to program a GDB debugging stub into flash which will then allow applications to be loaded into RAM via the serial port. This allows development to proceed without needing to use a JTAG device, although one will be required to program the debugging stub in the first place, and application size is limited to just 28KiB, since the GDB stub uses the least significant 4KiB.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-chip NOR flash is organized into 1024 pages of 256 bytes each.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports the Debug Unit and USART serial devices. The debug serial port at J2 can be used for communication. If the GDB stub ROM is installed, it uses the Debug Unit serial device. The serial driver package is loaded automatically when configuring for the AT91SAM7A3-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the AT91SAM7A3-EK target.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7A3-EK support is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM7A3-EK board for eCos Development

Overview

eCos applications are either programmed into the on-chip flash, or run from RAM using either a JTAG device or the GDB stubs ROM. To install a flash-resident application, or the GDB stubs requires use of a JTAG device to write to the flash. So, in all cases it is necessary to set up a JTAG device for the board. This document describes how to set up an Abatron BDI2000 and then use it to program an application into the flash.

Initial Installation

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.at91sam7a3ek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7a3ek/VERSION/misc` relative to the root of your eCos installation.
5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the `bdi2000.at91sam7a3ek.cfg` in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.
8. Open `bdi2000.at91sam7a3ek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91sam7a3ek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7A3-EK board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.
4. Power up the AT91SAM7A3-EK board.
5. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
SAM7A3>
```

6. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
SAM7A3> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x3F0F0F0F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
SAM7A3>
```

Installation into Flash

Installation of an application into the on-chip flash, or the installation of the GDB stubs take exactly the same form:

1. Locate the binary image of the executable to be installed. For the GDB stubs do this by locating the file `gdb.module.bin` within the `loaders` subdirectory of the base of the eCos installation. For applications use **arm-elf-objcopy -O binary** to convert the ELF output of the linker into binary.
2. Copy the file into a location on the host computer accessible to its TFTP server.
3. Connect to the BDI2000 telnet port as before.
4. Give the **unlock** command to ensure that the flash area we want to program is writable:

```
SAM7A3>unlock 0x100000 0x100 256
Unlocking flash at 0x00100000
Unlocking flash at 0x00100100
Unlocking flash at 0x00100200
...
Unlocking flash at 0x0010fe00
Unlocking flash at 0x0010ff00
Unlocking flash passed
```



```
SAM7A3>
```

This command unlocks 256 pages, i.e. 64KiB. The number of pages unlocked should match at least the size of the executable to be programmed.

5. Give the **erase** command to clear any previous contents:

```
SAM7A3>erase 0x100000 0x100 256
Erasing flash at 0x00100000
Erasing flash at 0x00100100
Erasing flash at 0x00100200
...
Erasing flash at 0x0010fe00
Erasing flash at 0x0010ff00
Erasing flash passed
SAM7A3>
```

As with the **unlock** command, the size of the area erased must be at least the size of the executable to be programmed.

6. Now give the **prog** command to fetch the executable from the TFTP server and program it to the flash.

```
SAM7A3>prog 0x100000 sam7.bin bin
Programming sam7.bin , please wait ....
Programming flash passed
SAM7A3>
```

The installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. A ROM based application should start immediately, and any output will be seen on the serial connection. If the GDB stub ROM has been installed, then something similar to the following will be seen on the serial port:

```
+$T050f:cc061000;0d:18082000;#4d
```


Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7A3-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7a3ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has GDB stubs programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x00100000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG

This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

GDB Stubs and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7A3-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AT91` package contains all the code and data definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7A3-EK board use the AT91SAM7A3's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Note that on the AT91, the on-chip watchdog peripheral always starts running immediately, and so in configurations that do not include the watchdog driver, it is always disabled via its write-once register. In configurations which include the watchdog driver obviously the watchdog is not disabled otherwise it could not be subsequently re-enabled, and so the application must start and periodically reset the watchdog from the very beginning of execution.

USART Serial Driver

The AT91SAM7A3-EK board use the AT91SAM7A3's internal USART serial support as described in the AT91 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and `"/dev/ser0"`. Only USART 0 supports modem control signals such as those used for hardware flow control.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7A3.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7A3 only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91sam7a3ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi2000.at91sam7a3ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the `bdiGDB` interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91sam7a3ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in **'reset halt'** mode (either with the **'reset halt'** or **'reset'** commands, or by pressing the reset button) and the **'go'** command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type **'go'** every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7A3>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7A3>go 0x00201000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi2000.at91sam7a3ek.cfg` configuration on the BDI2000, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB (\$O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 0 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7A3-EK hardware, and should be read in conjunction with that specification. The AT91SAM7A3-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

On-chip Flash

This is located at address 0x00100000 of the physical memory space.

On-chip RAM

This is located at address 0x00200000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x00201000, with the bottom 4KiB reserved for use by the GDB stubs.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used   456 size  3920
Startup : Interrupt stack used  4064 size  4096
Startup : Idlethread stack used   120 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 3 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 47.13 microseconds (141 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            1
Thread switches:    128
Mutexes:            32
Mailboxes:          21
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence			
Ave	Min	Max	Var	Ave	Min	Function	
=====	=====	=====	=====	=====	=====	=====	
144.33	144.33	144.33	0.00	100%	100%	Create thread	
18.33	18.33	18.33	0.00	100%	100%	Yield thread [all suspended]	
14.67	14.67	14.67	0.00	100%	100%	Suspend [suspended] thread	
14.67	14.67	14.67	0.00	100%	100%	Resume thread	
17.33	17.33	17.33	0.00	100%	100%	Set priority	
0.67	0.67	0.67	0.00	100%	100%	Get priority	
32.33	32.33	32.33	0.00	100%	100%	Kill [suspended] thread	
18.67	18.67	18.67	0.00	100%	100%	Yield [no other] thread	
22.67	22.67	22.67	0.00	100%	100%	Resume [suspended low prio] thread	
14.67	14.67	14.67	0.00	100%	100%	Resume [runnable low prio] thread	
18.00	18.00	18.00	0.00	100%	100%	Suspend [runnable] thread	
18.67	18.67	18.67	0.00	100%	100%	Yield [only low prio] thread	
14.67	14.67	14.67	0.00	100%	100%	Suspend [runnable->not runnable]	
32.00	32.00	32.00	0.00	100%	100%	Kill [runnable] thread	
31.33	31.33	31.33	0.00	100%	100%	Destroy [dead] thread	
55.67	55.67	55.67	0.00	100%	100%	Destroy [runnable] thread	
106.67	106.67	106.67	0.00	100%	100%	Resume [high priority] thread	
0.85	0.67	1.00	0.16	56%	43%	Scheduler lock	
13.31	13.00	13.33	0.04	93%	6%	Scheduler unlock [0 threads]	
13.31	13.00	13.33	0.04	93%	6%	Scheduler unlock [1 suspended]	
13.31	13.00	13.33	0.04	93%	6%	Scheduler unlock [many suspended]	

13.31	13.00	13.33	0.04	93%	6%	Scheduler unlock [many low prio]
2.33	2.33	2.33	0.00	100%	100%	Init mutex
18.58	18.33	18.67	0.13	75%	25%	Lock [unlocked] mutex
20.68	20.67	21.00	0.02	96%	96%	Unlock [locked] mutex
16.58	16.33	16.67	0.13	75%	25%	Trylock [unlocked] mutex
15.96	15.67	16.00	0.07	87%	12%	Trylock [locked] mutex
1.00	1.00	1.00	0.00	100%	100%	Destroy mutex
85.56	85.33	85.67	0.14	68%	31%	Unlock/Lock mutex
3.90	3.67	4.00	0.14	71%	28%	Create mbox
0.29	0.00	0.33	0.08	85%	14%	Peek [empty] mbox
19.90	19.67	20.00	0.14	71%	28%	Put [first] mbox
0.33	0.33	0.33	0.00	100%	100%	Peek [1 msg] mbox
19.92	19.67	20.00	0.12	76%	23%	Put [second] mbox
0.33	0.33	0.33	0.00	100%	100%	Peek [2 msgs] mbox
20.00	20.00	20.00	0.00	100%	100%	Get [first] mbox
20.00	20.00	20.00	0.00	100%	100%	Get [second] mbox
18.92	18.67	19.00	0.12	76%	23%	Tryput [first] mbox
16.49	16.33	16.67	0.17	52%	52%	Peek item [non-empty] mbox
17.49	17.33	17.67	0.17	52%	52%	Tryget [non-empty] mbox
16.33	16.33	16.33	0.00	100%	100%	Peek item [empty] mbox
16.56	16.33	16.67	0.15	66%	33%	Tryget [empty] mbox
0.40	0.33	0.67	0.10	80%	80%	Waiting to get mbox
0.40	0.33	0.67	0.10	80%	80%	Waiting to put mbox
2.16	2.00	2.33	0.17	52%	52%	Delete mbox
68.73	68.67	69.00	0.10	80%	80%	Put/Get mbox
2.33	2.33	2.33	0.00	100%	100%	Init semaphore
14.00	14.00	14.00	0.00	100%	100%	Post [0] semaphore
14.50	14.33	14.67	0.17	100%	50%	Wait [1] semaphore
13.96	13.67	14.00	0.07	87%	12%	Trywait [0] semaphore
14.00	14.00	14.00	0.00	100%	100%	Trywait [1] semaphore
2.38	2.33	2.67	0.07	87%	87%	Peek semaphore
0.83	0.67	1.00	0.17	100%	50%	Destroy semaphore
58.07	58.00	58.33	0.11	78%	78%	Post/Wait semaphore
4.00	4.00	4.00	0.00	100%	100%	Create counter
0.46	0.33	0.67	0.16	62%	62%	Get counter value
0.46	0.33	0.67	0.16	62%	62%	Set counter value
15.46	15.33	15.67	0.16	62%	62%	Tick counter
0.50	0.33	0.67	0.17	100%	50%	Delete counter
2.25	2.00	2.33	0.13	75%	25%	Init flag
14.33	14.33	14.33	0.00	100%	100%	Destroy flag
13.75	13.67	14.00	0.13	75%	75%	Mask bits in flag
15.67	15.67	15.67	0.00	100%	100%	Set bits in flag [no waiters]
19.83	19.67	20.00	0.17	100%	50%	Wait for flag [AND]
19.58	19.33	19.67	0.13	75%	25%	Wait for flag [OR]
19.83	19.67	20.00	0.17	100%	50%	Wait for flag [AND/CLR]
19.58	19.33	19.67	0.13	75%	25%	Wait for flag [OR/CLR]
0.25	0.00	0.33	0.13	75%	25%	Peek on flag
5.27	5.00	5.33	0.10	81%	18%	Create alarm
20.54	20.33	20.67	0.16	62%	37%	Initialize alarm

The HAL Port

```
13.90  13.67  14.00    0.14  68%  31% Disable alarm
19.85  19.67  20.00    0.16  56%  43% Enable alarm
14.65  14.33  14.67    0.04  93%   6% Delete alarm
17.42  17.33  17.67    0.13  75%  75% Tick counter [1 alarm]
82.65  82.33  82.67    0.04  93%   6% Tick counter [many alarms]
26.08  26.00  26.33    0.13  75%  75% Tick & fire counter [1 alarm]
371.00 369.67 402.67    1.98  96%  96% Tick & fire counters [>1 together]
91.63  91.33  91.67    0.07  87%  12% Tick & fire counters [>1 separately]
38.33  38.33  38.33    0.00 100% 100% Alarm latency [0 threads]
40.89  38.33  44.67    2.44  27%  47% Alarm latency [many threads]
93.75  93.67 104.67    0.17  99%  99% Alarm -> thread resume latency

    6.66    6.33    6.67    0.00                Clock/interrupt latency

14.37   13.00  261.33    0.00                Clock DSR latency

1360   1360   1360 (main stack: 3920) Thread stack used (1360 total)
      All done, main stack : stack used   896 size  3920
      All done : Interrupt stack used   212 size  4096
      All done : Idlethread stack used   304 size  2048
```

Timing complete - 23730 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM7A3-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXIII. Atmel AT91SAM7S-EK Board Support

Overview

Name

eCos Support for the Atmel AT91SAM7S-EK — Overview

Description

This document covers the configuration and usage of eCos and GDB Stubs on the Atmel AT91SAM7S-EK Evaluation Kit. The AT91SAM7S-EK Evaluation Kit contains the AT91SAM7S processor, external connections for two serial channels (one debug, one full), USB host/device. eCos support for the devices and peripherals on the AT91SAM7S is described below.

Application development on this board can take one of several approaches. Applications may be loaded into RAM via a JTAG device; however, the application size is limited by the amount of on-chip RAM, which is 64KiB on the AT91SAM7S256 and AT91SAM7S512, and 32KiB on the AT91SAM7S128. Applications may also be loaded into the on-chip flash memory where the RAM limit will only apply to the data portion of the application. Finally, it is possible to program a GDB debugging stub into flash which will then allow applications to be loaded into RAM via the serial port. This allows development to proceed without needing to use a JTAG device and application size is limited to the RAM size less 4KiB used by the stub. It is therefore recommended that JTAG debugging be used to debug applications since memory is limited on this platform.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-chip NOR flash is organized into pages of 128 or 256 bytes each. The number of pages is determined by the device variant, from 2048 for the AT91SAM7S512 to 128 for the AT91SAM7S32.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J3 and DTE port at J2 (connected to USART channel 0) can be used for communication. If the GDB stub ROM is installed, it uses the Debug Unit serial device only. The serial driver package is loaded automatically when configuring for the AT91SAM7S-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the AT91SAM7S-EK target.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7S-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM7S-EK board for eCos Development

Overview

eCos applications are either programmed into the on-chip flash, or run from RAM using either a JTAG device or the GDB stubs ROM. The installation of the GDB stubs or any flash-resident application requires use of a JTAG device to write to the flash, or the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. This document describes how to set up an Abatron BDI2000 (<http://www.abatron.ch>) Ronetix PEEDI (<http://www.ronetix.at/peedi.html>) for programming the gdb stubs and applications into the flash, and use the Atmel SAM-BA application to program the gdb stubs into the flash.

Initial Installation with Abatron BDI2000

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.at91sam7sek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7sek/VERSION/misc` relative to the root of your eCos installation.
5. Locate the file `regSAM7S.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the `bdi2000.at91sam7sek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `regSAM7S.def` in a location accessible to the TFTP server.
8. Open `bdi2000.at91sam7sek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `regSAM7S.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91sam7sek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7S-EK board for programming with BDI2000

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI2000.
4. Power up the AT91SAM7S-EK board.
5. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
SAM7S>
```

6. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
SAM7S> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x3F0F0F0F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
SAM7S>
```

Initial Installation with Ronetix PEEDI

Preparing the Ronetix PEEDI JTAG debugger

The PEEDI must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps give a typical outline of setting up the PEEDI using TFTP. Consult the PEEDI documentation for alternative mechanisms.

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Locate the file `peedi.at91sam7sek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7sek/VERSION/misc` relative to the root of your eCos installation.
4. Place the `peedi.at91sam7sek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.
5. Open `at91sam7sek.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.
6. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi.at91sam7sek.cfg` target configuration file on the TFTP server at the appropriate point of the **fconfig** process, for example with a path such as: `tftp://192.168.7.9/peedi.at91sam7sek.cfg`
7. Reset the PEEDI.
8. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet 192.168.7.225
Trying 192.168.7.225...
Connected to 192.168.7.225.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2007 www.ronetix.at - All rights reserved
Hw:1.2, Fw:2.0.13, SN: PD-0000-XXXX-XXXX
-----

sam7sek>
```

Preparing the AT91SAM7S-EK board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the PEEDI using a 20-pin ARM/Xscale cable from the JTAG interface connector on the board to the Target port on the PEEDI.
4. Power up the AT91SAM7S-EK board.
5. Connect to the PEEDI's CLI on port 23 as before.
6. Confirm correct connection with the PEEDI with the **reset** command as follows:

```
sam7sek> reset
++ info: user reset
sam7sek>
```

```

++ info: RESET and TRST asserted
++ info: TRST released
++ info: 1 TAP controller(s) detected
++ info: TAP : IDCODE = 0x3F0F0F0F, ARM7TDMI compliant
++ info: RESET released
++ info: core 0: initialized

sam7sek>

```

Installation into Flash

Installation of an application into the on-chip flash, or the installation of the GDB stubs, using a JTAG programmer takes exactly the same form:

1. Locate the binary image of the executable to be installed. For the GDB stubs do this by locating the file `gdb_module.bin` within the `loaders` subdirectory of the base of the eCos installation. For applications use **arm-eabi-objcopy -O binary** to convert the ELF output of the linker into binary.
2. Copy the file into a location on the host computer accessible to its TFTP server.
3. Connect to the JTAG device telnet port as before.
4. The flash must be unlocked to ensure that the flash area we want to program is writable.

For the BDI2000, use the **unlock** command:

```

SAM7S>unlock 0x100000 0x100 256
Unlocking flash at 0x00100000
Unlocking flash at 0x00100100
Unlocking flash at 0x00100200
...
Unlocking flash at 0x0010fe00
Unlocking flash at 0x0010ff00
Unlocking flash passed
SAM7S>

```

For the PEEDI, use the **flash unlock** command:

```

sam7sek> flash unlock 0x100000 65536
unlocking region #0 at 0x00100000
unlocking region #1 at 0x00104000
unlocking region #2 at 0x00108000
unlocking region #3 at 0x0010C000

sam7sek>

```

This command unlocks 256 pages, i.e. 64KiB on the AT91SAM7S512. The number of pages unlocked should match at least the size of the executable to be programmed. With the PEEDI you can use an unadorned **flash unlock** to unlock the entire flash.

5. Give the **erase** (BDI2000) or **flash erase** (PEEDI) command to clear any previous contents.

For the BDI2000:

```
SAM7S>erase 0x100000 0x100 256
Erasing flash at 0x00100000
Erasing flash at 0x00100100
Erasing flash at 0x00100200
...
Erasing flash at 0x0010fe00
Erasing flash at 0x0010ff00
Erasing flash passed
SAM7S>
```

As with the **unlock** command, the size of the area erased must be at least the size of the executable to be programmed.

For the PEEDI, only full flash erase is supported:

```
sam7sek> flash erase

done.

sam7sek>
```

6. Now give the **prog** (BDI2000) or **flash program** (PEEDI) command to fetch the executable from the TFTP server and program it to the flash.

For the BDI2000:

```
SAM7S>prog 0x100000 sam7.bin bin
Programming sam7.bin , please wait ....
Programming flash passed
SAM7S>
```

For the PEEDI:

```
sam7sek> flash program tftp://192.168.7.9/gdb_module.bin bin 0x100000
++ info: Programming directly
++ info: Programming image file: tftp://192.168.7.9/gdb_module.bin
++ info: At absolute address:      0x00100000
unlocking   at 0x00100000 (region #0)
programming at 0x00100000
programming at 0x00101000
programming at 0x00102000
programming at 0x00103000
unlocking   at 0x00104000 (region #1)
programming at 0x00104000
programming at 0x00105000
programming at 0x00106000
programming at 0x00107000
```

```
++ info: successfully programmed 32.00 KB in 0.93 sec

sam7sek>
```

The installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. A ROM based application should start immediately, and any output will be seen on the serial connection. If the GDB stub ROM has been installed, then something similar to the following will be seen on the serial port:

```
+$T050f:cc061000;0d:18082000;#4d
```

Programming GDB Stubs into Flash using SAM-BA

The following gives the steps needed to program the gdb stubs into Flash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=388). Install it on a suitable PC running Windows or Linux. The remainder of this section documents the behaviour seen under Windows, although the behaviour on Linux should not be too different.
2. Copy `gdb_module.bin` from either the `at91sam7sek_256` or `at91sam7sek_512` subdirectories, depending on which of the two boards you are using, to a suitable location on the PC.
3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 38400 baud. Connect a USB cable between the PC and the AT91SAM7S-EK board.
4. JP5 (TST) jumper needs to be temporarily closed and USB connected into the PC for 10 seconds. This writes the SAM-BA bootstrap into the boot sectors so the board can then be programmed. USB should then be disconnected and JP5 moved to the open position. If you do not do this then the option of connecting SAM-BA to the `\usb\ARM0` will not be available when the USB cable is reconnected to the PC.
5. Power up the board by plugging the USB cable from the AT91SAM7S-EK board into the PC and Windows should now recognize the USB device.
6. Start SAM-BA. Select "`\usb\ARM0`" for the communication interface, and "`at91sam7s256-ek`" or "`at91sam7s512-ek`" for the board to match the board you are about to program. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".
7. In the SAM-BA main window, select the "FLASH" tab and in the "Send File Name" field, select the `gdb_module.bin`. Ensure that the Address field contains "`0x100000`" and click "Send File". The following output should be seen:

```
(AT91-ISP v1.12) 1 % send_file {Flash} "gdb_module.bin" 0x100000 0
-I- Send File gdb_module.bin at address 0x100000
```

```

first_sector 0 last_sector 1
-I-      Writing: 0x740C bytes at 0x0 (buffer addr : 0x202B68)
-I-      0x740C bytes written by applet
(AT91-ISP v1.12) 1 %

```

You may get a popup asking if you want to unlock sectors 0, 1 of flash. Select "Y" if prompted.

You may also get a pop-up asking "Do you want to lock involved lock region(s) (0 to 1)?". Select "No" if prompted.

8. Shut down SAM-BA, disconnect and reconnect the USB cable. Press the reset button on the board and something similar to the following should be output for a AT91SAM7S256-EK board on the DEBUG serial line:

```
$T050f:c0051000;0d:e8072000;#7f
```

For a AT91SAM7S512-EK board you should see something similar. For example:

```
$T050f:c0051000;0d:e0072000;#44
```


Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7S-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7sek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has GDB stubs programmed into flash and boots into that initially. `arm-eabi-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x00100000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG

This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

GDB Stubs and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7S-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AT91` package contains all the code and data definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7S-EK board use the AT91SAM7S's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Note that on the AT91, the on-chip watchdog peripheral always starts running immediately, and so in configurations that do not include the watchdog driver, it is always disabled via its write-once register. In configurations which include the watchdog driver obviously the watchdog is not disabled otherwise it could not be subsequently re-enabled, and so the application must start and periodically reset the watchdog from the very beginning of execution.

USART Serial Driver

The AT91SAM7S-EK board use the AT91SAM7S's internal USART serial support as described in the AT91 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and `"/dev/ser0"`. Only USART 0 supports modem control signals such as those used for hardware flow control.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7S.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7S only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91sam7sek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi2000.at91sam7sek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the bdiGDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91sam7sek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7S>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7S>go 0x00201000
```

Consult the BDI2000 documentation for information on other formats.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam7sek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `peedi.at91sam7sek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb**. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam7sek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.at91sam7sek.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
sam7sek> memory load tftp://192.168.7.9/test.bin bin 0x201000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x00201000
loading at 0x201000
loading at 0x205000
```

```
Successfully loaded 28KB (29064 bytes) in 0.1s
sam7sek> go 0x201000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi2000.at91sam7sek.cfg` configuration on the BDI2000 or `peedi.at91sam7sek.cfg` configuration on the PEEDI, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB (\$O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 0 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7S-EK hardware, and should be read in conjunction with that specification. The AT91SAM7S-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

On-chip Flash

This is located at address 0x00100000 of the physical memory space.

On-chip RAM

This is located at address 0x00200000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x00201000, with the bottom 4KiB reserved for use by the GDB stubs.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    416 size  3920
Startup : Interrupt stack used    148 size  4096
Startup : Idlethread stack used     84 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 3 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 33.54 microseconds (100 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            2
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
22.00	21.67	22.33	0.33	100%	50%	Create thread
4.50	4.33	4.67	0.17	100%	50%	Yield thread [all suspended]
4.67	4.67	4.67	0.00	100%	100%	Suspend [suspended] thread
5.00	5.00	5.00	0.00	100%	100%	Resume thread
7.33	7.33	7.33	0.00	100%	100%	Set priority
0.67	0.67	0.67	0.00	100%	100%	Get priority
16.67	16.67	16.67	0.00	100%	100%	Kill [suspended] thread
4.67	4.67	4.67	0.00	100%	100%	Yield [no other] thread
8.67	8.33	9.00	0.33	100%	50%	Resume [suspended low prio] thread
5.00	5.00	5.00	0.00	100%	100%	Resume [runnable low prio] thread
6.50	6.33	6.67	0.17	100%	50%	Suspend [runnable] thread
4.50	4.33	4.67	0.17	100%	50%	Yield [only low prio] thread
4.33	4.33	4.33	0.00	100%	100%	Suspend [runnable->not runnable]
16.33	16.33	16.33	0.00	100%	100%	Kill [runnable] thread
11.67	11.67	11.67	0.00	100%	100%	Destroy [dead] thread
23.67	23.67	23.67	0.00	100%	100%	Destroy [runnable] thread
33.50	31.33	35.67	2.17	100%	50%	Resume [high priority] thread
12.62	12.33	17.33	0.14	74%	25%	Thread switch
0.44	0.33	0.67	0.14	68%	68%	Scheduler lock
3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [0 threads]
3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [1 suspended]

3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [many suspended]
3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [many low prio]
1.00	1.00	1.00	0.00	100%	100%	Init mutex
4.96	4.67	5.00	0.07	87%	12%	Lock [unlocked] mutex
5.84	5.67	6.00	0.17	53%	46%	Unlock [locked] mutex
4.83	4.67	5.00	0.17	100%	50%	Trylock [unlocked] mutex
4.13	4.00	4.33	0.16	62%	62%	Trylock [locked] mutex
0.50	0.33	0.67	0.17	100%	50%	Destroy mutex
32.27	32.00	32.33	0.10	81%	18%	Unlock/Lock mutex
1.46	1.33	1.67	0.16	62%	62%	Create mbox
0.33	0.33	0.33	0.00	100%	100%	Peek [empty] mbox
5.46	5.33	5.67	0.16	62%	62%	Put [first] mbox
0.29	0.00	0.33	0.07	87%	12%	Peek [1 msg] mbox
5.46	5.33	5.67	0.16	62%	62%	Put [second] mbox
0.29	0.00	0.33	0.07	87%	12%	Peek [2 msgs] mbox
5.58	5.33	5.67	0.13	75%	25%	Get [first] mbox
5.58	5.33	5.67	0.13	75%	25%	Get [second] mbox
4.63	4.33	4.67	0.07	87%	12%	Tryput [first] mbox
4.33	4.33	4.33	0.00	100%	100%	Peek item [non-empty] mbox
5.08	5.00	5.33	0.13	75%	75%	Tryget [non-empty] mbox
4.25	4.00	4.33	0.13	75%	25%	Peek item [empty] mbox
4.42	4.33	4.67	0.13	75%	75%	Tryget [empty] mbox
0.42	0.33	0.67	0.13	75%	75%	Waiting to get mbox
0.42	0.33	0.67	0.13	75%	75%	Waiting to put mbox
1.50	1.33	1.67	0.17	100%	50%	Delete mbox
22.02	22.00	22.33	0.04	93%	93%	Put/Get mbox
0.92	0.67	1.00	0.13	75%	25%	Init semaphore
4.00	4.00	4.00	0.00	100%	100%	Post [0] semaphore
4.54	4.33	4.67	0.16	62%	37%	Wait [1] semaphore
4.00	4.00	4.00	0.00	100%	100%	Trywait [0] semaphore
4.00	4.00	4.00	0.00	100%	100%	Trywait [1] semaphore
1.00	1.00	1.00	0.00	100%	100%	Peek semaphore
0.50	0.33	0.67	0.17	100%	50%	Destroy semaphore
19.92	19.67	20.00	0.13	75%	25%	Post/Wait semaphore
1.54	1.33	1.67	0.16	62%	37%	Create counter
0.46	0.33	0.67	0.16	62%	62%	Get counter value
0.46	0.33	0.67	0.16	62%	62%	Set counter value
4.92	4.67	5.00	0.13	75%	25%	Tick counter
0.50	0.33	0.67	0.17	100%	50%	Delete counter
0.88	0.67	1.00	0.16	62%	37%	Init flag
4.38	4.33	4.67	0.07	87%	87%	Destroy flag
4.00	4.00	4.00	0.00	100%	100%	Mask bits in flag
4.33	4.33	4.33	0.00	100%	100%	Set bits in flag [no waiters]
6.92	6.67	7.00	0.13	75%	25%	Wait for flag [AND]
6.79	6.67	7.00	0.16	62%	62%	Wait for flag [OR]
6.92	6.67	7.00	0.13	75%	25%	Wait for flag [AND/CLR]
6.83	6.67	7.00	0.17	100%	50%	Wait for flag [OR/CLR]
0.33	0.33	0.33	0.00	100%	100%	Peek on flag
2.67	2.67	2.67	0.00	100%	100%	Create alarm

```
8.63      8.33      8.67      0.07      87%   12% Initialize alarm
3.92      3.67      4.00      0.13      75%   25% Disable alarm
7.92      7.67      8.00      0.13      75%   25% Enable alarm
4.67      4.67      4.67      0.00     100%  100% Delete alarm
5.88      5.67      6.00      0.16      62%   37% Tick counter [1 alarm]
40.75     40.67     41.00      0.13      75%   75% Tick counter [many alarms]
11.54     11.33     11.67      0.16      62%   37% Tick & fire counter [1 alarm]
232.75    232.67    233.00      0.13      75%   75% Tick & fire counters [>1 together]
46.75     46.67     47.00      0.13      75%   75% Tick & fire counters [>1 separately]
32.33     32.33     32.33      0.00     100%  100% Alarm latency [0 threads]
35.22     33.33     39.67      1.90      78%   78% Alarm latency [2 threads]
35.22     32.33     39.67      1.90      77%   52% Alarm latency [many threads]
54.37     54.33     59.00      0.07      99%   99% Alarm -> thread resume latency

6.65      6.33      7.00      0.00                                Clock/interrupt latency

12.95     11.33     19.67      0.00                                Clock DSR latency

292      292      292 (main stack: 772) Thread stack used (1360 total)
      All done, main stack : stack used 772 size 3920
      All done : Interrupt stack used 208 size 4096
      All done : Idlethread stack used 248 size 2048
```

Timing complete - 30250 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM7S-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXIV. Atmel AT91SAM7X-EK Board Support

Overview

Name

eCos Support for the Atmel AT91SAM7X-EK — Overview

Description

This document covers the configuration and usage of eCos and GDB Stubs on the Atmel AT91SAM7X-EK Evaluation Kit. The AT91SAM7X-EK Evaluation Kit contains the AT91SAM7X256 processor, external connections for two serial channels (one debug, one full), ethernet, USB host/device. eCos support for the devices and peripherals on the AT91SAM7X256 is described below.

Application development on this board can take one of several approaches. Applications may be loaded into RAM via a JTAG device; however, the application size is limited by the amount of on-chip RAM, which is 64KiB on the AT91SAM7X256, 32KiB on the AT91SAM7X128 and 128KiB on the AT91SAM7X512. Applications may also be loaded into the on-chip flash memory where the RAM limit will only apply to the data portion of the application. Finally, it is possible to program a GDB debugging stub into flash which will then allow applications to be loaded into RAM via the serial port. This allows development to proceed without needing to use a JTAG device and application size is limited to the RAM size less 4KiB used by the stub. It is therefore recommended that JTAG debugging be used to debug applications since memory is limited on this platform.

This documentation is expected to be read in conjunction with the AT91 processor HAL and AT91SAM7 variant HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-chip NOR flash is organized into pages of 256 bytes each. The number of pages is determined by the device variant, from 512 for the AT91SAM7X128 to 2048 for the AT91SAM7X512.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J3 and DTE port at J2 (connected to USART channel 0) can be used for communication. If the GDB stub ROM is installed, it uses the Debug Unit serial device only. The serial driver package is loaded automatically when configuring for the AT91SAM7X-EK target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the AT91SAM7X-EK target.

The AT91SAM7 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM7X. This type of bus is also known as I²C®.

There is a network driver for the on-chip Ethernet and DM9161A PHY. This is only recommended for use with the lwIP TCP/IP stack due to the low RAM requirements.

In general, devices (PIO, UARTs, etc.) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM7X-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM7X-EK board for eCos Development

Overview

eCos applications are either programmed into the on-chip flash, or run from RAM using either a JTAG device or the GDB stubs ROM. The installation of the GDB stubs or any flash-resident application requires use of a JTAG device to write to the flash, or the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. So, in all cases it is necessary to set up a JTAG device for the board. This document describes how to set up either an Abatron BDI3000 (<http://www.abatron.ch>) or Ronetix PEEDI (<http://www.ronetix.at/peedi.html>) and then use them to program an application into the flash.

Initial Installation with Abatron BDI3000

Preparing the Abatron BDI3000 JTAG debugger

The BDI3000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI3000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI3000.
3. Install the Abatron BDI3000 bdiGDB support software on the host PC.
4. Locate the file `bdi3000.at91sam7xek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7xek/VERSION/misc` relative to the root of your eCos installation.
5. Locate the file `regSAM7S.def` within the installation of the BDI3000 bdiGDB support software.
6. Place the `bdi3000.at91sam7xek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the BDI3000 to load this file via TFTP as its configuration file.
7. Similarly place the file `regSAM7S.def` in a location accessible to the TFTP server.
8. Open `bdi3000.at91sam7xek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `regSAM7S.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI3000 in line with the bdiGDB instruction manual. Configure the BDI3000 to use the `bdi3000.at91sam7xek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91SAM7X-EK board for programming with BDI3000

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the BDI3000 using a 20-pin ARM/Xscale cable from the JTAG interface connector to the Target A port on the BDI3000.
4. Power up the AT91SAM7X-EK board.
5. Connect to the BDI3000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
SAM7X>
```

6. Confirm correct connection with the BDI3000 with the **reset halt** command as follows:

```
SAM7X> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x3F0F0F0F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
SAM7X>
```

Initial Installation with Ronetix PEEDI

Preparing the Ronetix PEEDI JTAG debugger

The PEEDI must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps give a typical outline of setting up the PEEDI using TFTP. Consult the PEEDI documentation for alternative mechanisms.

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the PEEDI JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the PEEDI (straight through, not null modem).

3. Locate the file `peedi.at91sam7xek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/at91/at91sam7xek/VERSION/misc` relative to the root of your eCos installation.
4. Place the `peedi.at91sam7xek.cfg` file in a location on the PC accessible to the TFTP server. Later you will configure the PEEDI to load this file via TFTP as its configuration file.
5. Open `at91sam7xek.cfg` in an editor such as emacs or notepad and insert your own license information in the `[LICENSE]` section.
6. Install and configure the PEEDI in line with the PEEDI Quick Start Guide or User's Manual, especially configuring PEEDI's RedBoot with the network information. Configure it to use the `peedi.at91sam7xek.cfg` target configuration file on the TFTP server at the appropriate point of the **fconfig** process, for example with a path such as: `tftp://192.168.7.9/peedi.at91sam7xek.cfg`
7. Reset the PEEDI.
8. Connect to the PEEDI's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see output similar to the following:

```
$ telnet 192.168.7.225
Trying 192.168.7.225...
Connected to 192.168.7.225.
Escape character is '^]'.

PEEDI - Powerful Embedded Ethernet Debug Interface
Copyright (c) 2005-2007 www.ronetix.at - All rights reserved
Hw:1.2, Fw:2.0.13, SN: PD-0000-XXXX-XXXX
-----

sam7xek>
```

Preparing the AT91SAM7X-EK board for programming with PEEDI

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 38400 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to the PEEDI using a 20-pin ARM/Xscale cable from the JTAG interface connector on the board to the Target port on the PEEDI.
4. Power up the AT91SAM7X-EK board.
5. Connect to the PEEDI's CLI on port 23 as before.
6. Confirm correct connection with the PEEDI with the **reset** command as follows:

```
sam7xek> reset
++ info: user reset
sam7xek>
```

```
++ info: RESET and TRST asserted
++ info: TRST released
++ info: 1 TAP controller(s) detected
++ info: TAP : IDCODE = 0x3F0F0F0F, ARM7TDMI compliant
++ info: RESET released
++ info: core 0: initialized

sam7xek>
```

Installation into Flash

Installation of an application into the on-chip flash, or the installation of the GDB stubs, using a JTAG programmer takes exactly the same form:

1. Locate the binary image of the executable to be installed. For the GDB stubs do this by locating the file `gdb.module.bin` within the `loaders` subdirectory of the base of the eCos installation. For applications use **arm-elf-objcopy -O binary** to convert the ELF output of the linker into binary.
2. Copy the file into a location on the host computer accessible to its TFTP server.
3. Connect to the JTAG device telnet port as before.
4. The flash must be unlocked to ensure that the flash area we want to program is writable.

For the BDI3000, use the **unlock** command:

```
SAM7X>unlock 0x100000 0x100 256
Unlocking flash at 0x00100000
Unlocking flash at 0x00100100
Unlocking flash at 0x00100200
...
Unlocking flash at 0x0010fe00
Unlocking flash at 0x0010ff00
Unlocking flash passed
SAM7X>
```

For the PEEDI, use the **flash unlock** command:

```
sam7xek> flash unlock 0x100000 65536
unlocking region #0 at 0x00100000
unlocking region #1 at 0x00104000
unlocking region #2 at 0x00108000
unlocking region #3 at 0x0010C000

sam7xek>
```

This command unlocks 256 pages, i.e. 64KiB. The number of pages unlocked should match at least the size of the executable to be programmed. With the PEEDI you can use an unadorned **flash unlock** to unlock the entire flash.

5. Give the **erase** (BDI3000) or **flash erase** (PEEDI) command to clear any previous contents.

For the BDI3000:

```
SAM7X>erase 0x100000 0x100 256
Erasing flash at 0x00100000
Erasing flash at 0x00100100
Erasing flash at 0x00100200
...
Erasing flash at 0x0010fe00
Erasing flash at 0x0010ff00
Erasing flash passed
SAM7X>
```

As with the **unlock** command, the size of the area erased must be at least the size of the executable to be programmed.

For the PEEDI, only full flash erase is supported:

```
sam7xek> flash erase

done.

sam7xek>
```

6. Now give the **prog** (BDI3000) or **flash program** (PEEDI) command to fetch the executable from the TFTP server and program it to the flash.

For the BDI3000:

```
SAM7X>prog 0x100000 sam7.bin bin
Programming sam7.bin , please wait ....
Programming flash passed
SAM7X>
```

For the PEEDI:

```
sam7xek> flash program tftp://192.168.7.9/gdb_module.bin bin 0x100000
++ info: Programming directly
++ info: Programming image file: tftp://192.168.7.9/gdb_module.bin
++ info: At absolute address:      0x00100000
unlocking   at 0x00100000 (region #0)
programming at 0x00100000
programming at 0x00101000
programming at 0x00102000
programming at 0x00103000
unlocking   at 0x00104000 (region #1)
programming at 0x00104000
programming at 0x00105000
programming at 0x00106000
programming at 0x00107000
```

```
++ info: successfully programmed 32.00 KB in 0.93 sec

sam7xek>
```

7. Finally, the processor must be switched to boot from the flash rather than the internal ROM. This is done by programming a General Purpose NVM bit in the flash memory. This can be done using memory write commands in the JTAG device telnet interface. On the BDI3000 this is done using the following commands:

```
SAM7X>md 0xffffffff60 3
ffffffff60 : 0x00300100      3145984  ..0.
ffffffff64 : 0x00000000      0       ....
ffffffff68 : 0x00000001      1       ....
SAM7X>mm 0xffffffff64 0x5a00020b
SAM7X>md 0xffffffff60 3
ffffffff60 : 0x00300100      3145984  ..0.
ffffffff64 : 0x00000000      0       ....
ffffffff68 : 0x00000401      1025    ....
SAM7X>
```

And on the PEEDI:

```
sam7xek> mem read 0xffffffff60 3

0xFFFFFFFF60: 0x00300100 0x00000000 0x00000001
sam7xek> mem write 0xffffffff64 0x5a00020b
sam7xek> mem read 0xffffffff60 3

0xFFFFFFFF60: 0x00300100 0x00000000 0x00000401
sam7xek>
```

The installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. A ROM based application should start immediately, and any output will be seen on the serial connection. If the GDB stub ROM has been installed, then something similar to the following will be seen on the serial port:

```
+$T050f:cc061000;0d:18082000;#4d
```

Programming GDB Stubs into Flash using SAM-BA

The following gives the steps needed to program the gdb stubs into Flash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=388). Install it on a suitable PC running

Windows or Linux. The remainder of this section documents the behaviour seen under Windows, although the behaviour on Linux should not be too different.

2. Copy `gdb_module.bin` from either the `at91sam7xek_256` or `at91sam7xek_512` subdirectories, depending on which of the two boards you are using, to a suitable location on the PC.
3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 38400 baud. Connect a USB cable between the PC and the AT91SAM7X-EK board.
4. Power up the board by plugging the USB cable from the AT91SAM7X-EK board into the PC and Windows should recognize the USB device. If it does not, then you will need to erase the existing program that has already been programmed into flash. To do this, disconnect the USB cable from the PC, effectively powering down the device, connect jumper J8 (ERASE) on the AT91SAM7X-EK board and reconnect the USB cable. This step should have erased the flash. Finally disconnect the USB cable followed by J8, reconnect the USB cable and the board should be recognized now. Windows may ask you to install a new driver, in which case follow the instructions.
5. Start SAM-BA. Select "\\usb\\ARM0" for the communication interface, and "at91sam7x256-ek" or "at91sam7x512-ek" for the board to match the board you are about to program. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".

6. In the SAM-BA main window, select the "FLASH" tab and in the "Send File Name" field, select the `gdb_module.bin`. Ensure that the Address field contains "0x100000" and click "Send File". The following output should be seen:

```
(AT91-ISP v1.13) 1 % send_file {Flash} "gdb_module.bin" 0x100000 0
-I- Send File gdb_module.bin at address 0x100000
    first_sector 0 last_sector 1
-I- Writing: 0x7400 bytes at 0x0 (buffer addr : 0x202BC8)
-I- 0x7400 bytes written by applet
(AT91-ISP v1.13) 1 %
```

You may get a pop-up asking "Do you want to lock involved lock region(s) (0 to 1)?" Select "No" if prompted.

7. In the Scripts section, select the script "Boot from Flash (GPNVM2)" and press "Execute". The following output should be seen:

```
(AT91-ISP v1.13) 1 % FLASH::ScriptGPNMV 4
-I- GPNVM2 set
(AT91-ISP v1.13) 1 %
```

8. Shut down SAM-BA, disconnect and reconnect the USB cable. Press the reset button on the board and something similar to the following should be output for a AT91SAM7X256-EK board on the DEBUG serial line:

```
$T050f:cc051000;0d:e8072000;#7f
```

For a AT91SAM7X512-EK board you should see something similar. For example:

```
$T050f:d0051000;0d:e8072000;#4d
```


Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM7X-EK platform HAL package is loaded automatically when eCos is configured for the `at91sam7xek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has GDB stubs programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by the stubs. By default the application will use the eCos virtual vectors mechanism to obtain certain services from the stubs, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x00100000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

JTAG

This is an alternative development startup type. The application is loaded into RAM via a JTAG device and is run and debugged from there. The application will be self-contained with no dependencies on services provided by other software. It is expected that hardware setup will have been performed via the JTAG device prior to loading.

GDB Stubs and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building the GDB Stubs.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM7X-EK board contains a quantity of on-chip flash memory. The `CYGPKG_DEVS_FLASH_AT91` package contains all the code and data definitions necessary to support this part. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Watchdog Driver

The AT91SAM7X-EK board uses the AT91SAM7X's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Note that on the AT91, the on-chip watchdog peripheral always starts running immediately, and so in configurations that do not include the watchdog driver, it is always disabled via its write-once register. In configurations which include the watchdog driver obviously the watchdog is not disabled otherwise it could not be subsequently re-enabled, and so the application must start and periodically reset the watchdog from the very beginning of execution.

USART Serial Driver

The AT91SAM7X-EK board use the AT91SAM7X's internal USART serial support as described in the AT91 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and `"/dev/ser0"`. Only USART 0 supports modem control signals such as those used for hardware flow control.

Ethernet Driver

The AT91SAM7X-EK board can use the AT91SAM7X's internal ethernet MAC (EMAC) support. The package `CYGPKG_DEVS_ETH_ARM_AT91` contains the necessary device driver support.

Due to the amount of RAM available, it is not possible to use the BSD-derived TCP/IP stack. Instead either the lwIP TCP/IP stack (in `CYGPKG_NET_LWIP`) should be used, or your application can operate the driver directly in standalone mode. To enable the Ethernet driver in your configuration, either include the "Common Ethernet Support" (`CYGPKG_IO_ETH_DRIVERS`) package in your configuration, or base your configuration on an appropriate template, in particular the `"lwip_eth"` template.

Support for the Davicom DM9161A PHY (which comes from the `CYGPKG_DEVS_ETH_PHY` package) is automatically configured when ethernet support is enabled.

The AT91 ethernet device driver package in fact includes two separate driver implementations: one standard driver suitable for use in standalone mode, or with various TCP/IP stacks including at least RedBoot, BSD and lwIP;

and one specific to lwIP. It is strongly recommended that the lwIP-specific driver is used with lwIP, given the low memory constraints. The lwIP-specific driver is a streamlined efficient version designed for very low RAM overhead. As a result it is implemented intentionally at the expense of features irrelevant to the AT91SAM7X, such as multiple network device support, and network debugging under RedBoot. Instead it has improvements such as zero-copy reception and transmission of data packets, leading to both faster operation, and smaller code and data memory footprint.

The choice between using the standard driver, or the lwIP-specific driver is not made within this package, but is instead made in the generic ethernet I/O package `CYGPKG_IO_ETH_DRIVERS` using the options within the lwIP driver model component (`CYGIMP_IO_ETH_DRIVERS_LWIP_DRIVER_MODEL`). The standard driver is the default, and thus needs to be explicitly changed to select the lwIP-specific driver model.

Careful selection and tuning of the configuration settings within the AT91 Ethernet package, and more especially, the lwIP stack, can result in realistic application use of TCP/IP at high speeds, despite the low RAM availability.

Note that a board design issue has required a workaround to be used in order to correctly initialize the PHY. This has two notable consequences: there is an extra delay of 700ms at startup time; and the programmatic use of the NRST line from the processor can confuse attached JTAG units into believing the processor has reset, which can in turn adversely affect debugging sessions. In the case of the Abatron BDI3000, it has been found that ensuring the last reset command was a "reset run", and then connecting via GDB immediately after target reset results in a stable debug session, albeit at the expense of the processor already having run some of the early HAL startup sequence.

Compiler Flags

The SAM7 variant HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm7tdmi`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm7tdmi` is the correct option for the ARM7TDMI processor in the SAM7X.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM7TDMI core of the AT91SAM7X only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI3000 notes

On the Abatron BDI3000, the `bdi3000.at91sam7xek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `bdi3000.at91sam7xek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI3000 command line interface to make the changes take effect.

On the BDI3000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the `bdiGDB` interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI3000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI3000 is powered up, the target will always run the initialization section of the `bdi3000.at91sam7xek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot from ROM as normal.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
SAM7X>load 0x00201000 /test.bin bin
Loading /test.bin , please wait ....
Loading program file passed
SAM7X>go 0x00201000
```

Consult the BDI3000 documentation for information on other formats.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam7xek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the PLL and flash memory controller.

The `peedi.at91sam7xek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[TARGET]` section. The supplied version of the file defaults to software breakpoints. With this default, hardware breakpoints can still be set from GDB using the **hbbreak** command. The default can be changed to hardware breakpoints, and remember to use the **reboot** command on the PEEDI command line interface, or press the reset button to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-elf-gdb**. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam7xek.cfg` file (which configures the CPU clock among other things), and halts the target. This behaviour is repeated with the **reset** command.

If the board is reset with the '**reset**' command, or by pressing the reset button and the '**go**' command is then given, then the board will boot from ROM as normal. A similar effect can be achieved in GDB by connecting with **target remote** and immediately typing **continue** or **c**.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the `CORE0_STARTUP_MODE` directive in the `[TARGET]` section of the `peedi.at91sam7xek.cfg` file. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
sam7xek> memory load tftp://192.168.7.9/test.bin bin 0x201000
++ info: Loading image file:  tftp://192.168.7.9/test.bin
++ info: At absolute address: 0x00201000
loading at 0x201000
loading at 0x205000
```

```
Successfully loaded 28KB (29064 bytes) in 0.1s
sam7xek> go 0x201000
```

Consult the PEEDI documentation for information on other formats and loading mechanisms.

Configuration of RAM applications

If the JTAG device has initialized the processor, such as by using the `bdi3000.at91sam7xek.cfg` configuration on the BDI3000 or `peedi.at91sam7xek.cfg` configuration on the PEEDI, applications can be loaded directly into RAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB (\$O) packets. Selecting the JTAG startup type in the configuration tool sets these options automatically.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 0 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM7X-EK hardware, and should be read in conjunction with that specification. The AT91SAM7X-EK platform HAL package complements the ARM architectural HAL, the AT91 variant HAL and the AT91SAM7 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor or JTAG device for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the PLL and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The AT91SAM7 processor HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

On-chip Flash

This is located at address 0x00100000 of the physical memory space.

On-chip RAM

This is located at address 0x00200000 of the physical memory space. During booting this memory is only available at this address, but during the boot process it is also remapped to location 0x00000000 in order to allow the hardware exception vectors to be in RAM. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM startup, all remaining RAM is available. For RAM startup, available RAM starts at location 0x00201000, with the bottom 4KiB reserved for use by the GDB stubs.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used 416 size 3920
Startup : Interrupt stack used 148 size 4096
Startup : Idlethread stack used 84 size 2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 3 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 33.54 microseconds (100 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            2
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
22.00	21.67	22.33	0.33	100%	50%	Create thread
4.50	4.33	4.67	0.17	100%	50%	Yield thread [all suspended]
4.67	4.67	4.67	0.00	100%	100%	Suspend [suspended] thread
5.00	5.00	5.00	0.00	100%	100%	Resume thread
7.33	7.33	7.33	0.00	100%	100%	Set priority
0.67	0.67	0.67	0.00	100%	100%	Get priority
16.67	16.67	16.67	0.00	100%	100%	Kill [suspended] thread
4.67	4.67	4.67	0.00	100%	100%	Yield [no other] thread
8.67	8.33	9.00	0.33	100%	50%	Resume [suspended low prio] thread
5.00	5.00	5.00	0.00	100%	100%	Resume [runnable low prio] thread
6.50	6.33	6.67	0.17	100%	50%	Suspend [runnable] thread
4.50	4.33	4.67	0.17	100%	50%	Yield [only low prio] thread
4.33	4.33	4.33	0.00	100%	100%	Suspend [runnable->not runnable]
16.33	16.33	16.33	0.00	100%	100%	Kill [runnable] thread
11.67	11.67	11.67	0.00	100%	100%	Destroy [dead] thread
23.67	23.67	23.67	0.00	100%	100%	Destroy [runnable] thread
33.50	31.33	35.67	2.17	100%	50%	Resume [high priority] thread
12.62	12.33	17.33	0.14	74%	25%	Thread switch
0.44	0.33	0.67	0.14	68%	68%	Scheduler lock
3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [0 threads]
3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [1 suspended]

3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [many suspended]
3.35	3.33	3.67	0.04	93%	93%	Scheduler unlock [many low prio]
1.00	1.00	1.00	0.00	100%	100%	Init mutex
4.96	4.67	5.00	0.07	87%	12%	Lock [unlocked] mutex
5.84	5.67	6.00	0.17	53%	46%	Unlock [locked] mutex
4.83	4.67	5.00	0.17	100%	50%	Trylock [unlocked] mutex
4.13	4.00	4.33	0.16	62%	62%	Trylock [locked] mutex
0.50	0.33	0.67	0.17	100%	50%	Destroy mutex
32.27	32.00	32.33	0.10	81%	18%	Unlock/Lock mutex
1.46	1.33	1.67	0.16	62%	62%	Create mbox
0.33	0.33	0.33	0.00	100%	100%	Peek [empty] mbox
5.46	5.33	5.67	0.16	62%	62%	Put [first] mbox
0.29	0.00	0.33	0.07	87%	12%	Peek [1 msg] mbox
5.46	5.33	5.67	0.16	62%	62%	Put [second] mbox
0.29	0.00	0.33	0.07	87%	12%	Peek [2 msgs] mbox
5.58	5.33	5.67	0.13	75%	25%	Get [first] mbox
5.58	5.33	5.67	0.13	75%	25%	Get [second] mbox
4.63	4.33	4.67	0.07	87%	12%	Tryput [first] mbox
4.33	4.33	4.33	0.00	100%	100%	Peek item [non-empty] mbox
5.08	5.00	5.33	0.13	75%	75%	Tryget [non-empty] mbox
4.25	4.00	4.33	0.13	75%	25%	Peek item [empty] mbox
4.42	4.33	4.67	0.13	75%	75%	Tryget [empty] mbox
0.42	0.33	0.67	0.13	75%	75%	Waiting to get mbox
0.42	0.33	0.67	0.13	75%	75%	Waiting to put mbox
1.50	1.33	1.67	0.17	100%	50%	Delete mbox
22.02	22.00	22.33	0.04	93%	93%	Put/Get mbox
0.92	0.67	1.00	0.13	75%	25%	Init semaphore
4.00	4.00	4.00	0.00	100%	100%	Post [0] semaphore
4.54	4.33	4.67	0.16	62%	37%	Wait [1] semaphore
4.00	4.00	4.00	0.00	100%	100%	Trywait [0] semaphore
4.00	4.00	4.00	0.00	100%	100%	Trywait [1] semaphore
1.00	1.00	1.00	0.00	100%	100%	Peek semaphore
0.50	0.33	0.67	0.17	100%	50%	Destroy semaphore
19.92	19.67	20.00	0.13	75%	25%	Post/Wait semaphore
1.54	1.33	1.67	0.16	62%	37%	Create counter
0.46	0.33	0.67	0.16	62%	62%	Get counter value
0.46	0.33	0.67	0.16	62%	62%	Set counter value
4.92	4.67	5.00	0.13	75%	25%	Tick counter
0.50	0.33	0.67	0.17	100%	50%	Delete counter
0.88	0.67	1.00	0.16	62%	37%	Init flag
4.38	4.33	4.67	0.07	87%	87%	Destroy flag
4.00	4.00	4.00	0.00	100%	100%	Mask bits in flag
4.33	4.33	4.33	0.00	100%	100%	Set bits in flag [no waiters]
6.92	6.67	7.00	0.13	75%	25%	Wait for flag [AND]
6.79	6.67	7.00	0.16	62%	62%	Wait for flag [OR]
6.92	6.67	7.00	0.13	75%	25%	Wait for flag [AND/CLR]
6.83	6.67	7.00	0.17	100%	50%	Wait for flag [OR/CLR]
0.33	0.33	0.33	0.00	100%	100%	Peek on flag
2.67	2.67	2.67	0.00	100%	100%	Create alarm

```
8.63      8.33      8.67      0.07      87%  12% Initialize alarm
3.92      3.67      4.00      0.13      75%  25% Disable alarm
7.92      7.67      8.00      0.13      75%  25% Enable alarm
4.67      4.67      4.67      0.00     100% 100% Delete alarm
5.88      5.67      6.00      0.16      62%  37% Tick counter [1 alarm]
40.75     40.67     41.00      0.13      75%  75% Tick counter [many alarms]
11.54     11.33     11.67      0.16      62%  37% Tick & fire counter [1 alarm]
232.75    232.67    233.00      0.13      75%  75% Tick & fire counters [>1 together]
46.75     46.67     47.00      0.13      75%  75% Tick & fire counters [>1 separately]
32.33     32.33     32.33      0.00     100% 100% Alarm latency [0 threads]
35.22     33.33     39.67      1.90      78%  78% Alarm latency [2 threads]
35.22     32.33     39.67      1.90      77%  52% Alarm latency [many threads]
54.37     54.33     59.00      0.07      99%  99% Alarm -> thread resume latency

6.65      6.33      7.00      0.00                      Clock/interrupt latency

12.95     11.33     19.67      0.00                      Clock DSR latency

292      292      292 (main stack: 772) Thread stack used (1360 total)
      All done, main stack : stack used 772 size 3920
      All done : Interrupt stack used 208 size 4096
      All done : Idlethread stack used 248 size 2048
```

Timing complete - 30250 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM7X-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91SAM7 processor HAL, AT91 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXV. Atmel SAM9 Processor Support

Overview

Name

Support for the Atmel SAM9 Processor — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91SAM9XXX processor family. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, ARM9 variant HAL and the platform HAL. It provides functionality common to all SAM9-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/arm9/sam9` within the eCos source repository.

The SAM9 processor HAL package is loaded automatically when eCos is configured for an SAM9-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel SAM9 processor within this processor HAL package include:

- [SAM9-specific hardware definitions](#)
- [Interrupt controller](#)
- [Timer counters](#)
- [Serial UARTs](#)
- [MultiMedia Card Interface \(MCI\)](#)
- [Two-Wire Interface \(TWI\)](#)
- [Power saving](#)

Support for the on-chip ethernet(AT91SAM9260 only), interrupt-driven serial, SPI, watchdog and wallclock (RTTC) features of the SAM9 are also present and can be found in separate packages, outside of this processor HAL.

Hardware definitions

Name

SAM9 hardware definitions — Details on obtaining hardware definitions for SAM9

Register definitions

The file `<cyg/hal/sam9.h>` can be included from application and eCos package sources to provide definitions related to SAM9 subsystems. These include register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, TWI (I²C®), Ethernet, timer counter, RTTC, and SPI subsystems.

Initialization helper macros

The file `<cyg/hal/sam9_init.inc>` contains definitions of helper macros which may be used by SAM9 platform HALs in order to initialise common SAM9 subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary.

Interrupt controller

Name

SAM9 interrupt controller — Advanced Interrupt Controller definitions and usage

Interrupt controller definitions

The file <cyg/hal/var_ints.h> (located at hal/arm/arm9/sam9/*VERSION*/include/var_ints.h in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs:

```
// The following are common to all SAM9 devices. Per-variant
// variations are supplied later.

#define CYGNUM_HAL_INTERRUPT_FIQ      0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYSTEM  1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA     2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB     3 // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_PIOC     4 // Parallel IO Controller C
                                        // Vector 5 variant specific
#define CYGNUM_HAL_INTERRUPT_US0      6 // USART 0
#define CYGNUM_HAL_INTERRUPT_US1      7 // USART 1
#define CYGNUM_HAL_INTERRUPT_US2      8 // USART 2
#define CYGNUM_HAL_INTERRUPT_MCI      9 // Multimedia Card Interface
#define CYGNUM_HAL_INTERRUPT_UDP     10 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TWI     11 // Two-Wire Interface
#define CYGNUM_HAL_INTERRUPT_SPI     12 // Serial Peripheral Interface 0
#define CYGNUM_HAL_INTERRUPT_SPI1    13 // Serial Peripheral Interface 1
#define CYGNUM_HAL_INTERRUPT_SSC0    14 // Serial Synchronous Controller 0
                                        // Vector 15 variant specific
                                        // Vector 16 variant specific
#define CYGNUM_HAL_INTERRUPT_TC0     17 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1     18 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2     19 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_UHP     20 // USB Host port
                                        // Vectors 21..28 variant specific
#define CYGNUM_HAL_INTERRUPT_IRQ0    29 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_INTERRUPT_IRQ1    30 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_INTERRUPT_IRQ2    31 // Advanced Interrupt Controller (IRQ2)

// Variant specific vectors

#if defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9260)

#define CYGNUM_HAL_INTERRUPT_ADC      5 // Analog to Digital Converter
                                        // Vector 15 unused
                                        // Vector 16 unused
#define CYGNUM_HAL_INTERRUPT_EMAC    21 // Ethernet MAC
#define CYGNUM_HAL_INTERRUPT_ISI     22 // Image Sensor Interface
#define CYGNUM_HAL_INTERRUPT_US3     23 // USART 3
#define CYGNUM_HAL_INTERRUPT_US4     24 // USART 4
```

```

#define CYGNUM_HAL_INTERRUPT_US5      25 // USART 5
#define CYGNUM_HAL_INTERRUPT_TC3      26 // Timer Counter 3
#define CYGNUM_HAL_INTERRUPT_TC4      27 // Timer Counter 4
#define CYGNUM_HAL_INTERRUPT_TC5      28 // Timer Counter 5

#elif defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9261)

// Vector 5 unused
#define CYGNUM_HAL_INTERRUPT_SSC1     15 // Serial Synchronous Controller 1
#define CYGNUM_HAL_INTERRUPT_SSC2     16 // Serial Synchronous Controller 2
#define CYGNUM_HAL_INTERRUPT_LCD      21 // LCD controller
// Vectors 22..28 unused

#else
#error Unknown SAM9 variant
#endif

// The following interrupts are derived from the SYSTEM interrupt
#define CYGNUM_HAL_INTERRUPT_PITC     32 // System Timer Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_DEBUG    33 // Debug unit
#define CYGNUM_HAL_INTERRUPT_WDTC     34 // Watchdog
#define CYGNUM_HAL_INTERRUPT_RTTC     35 // Real Time Clock
#define CYGNUM_HAL_INTERRUPT_PMC      36 // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_RSTC     37 // Reset Controller

```

As indicated above, further decoding is performed on the SYSTEM interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the SYSTEM interrupt will not work as expected. Conversely, masking a decoded derivative of the SYSTEM interrupt will not work as this would mask other SYSTEM interrupts, but masking the SYSTEM interrupt itself will work. On the other hand, unmasking a decoded SYSTEM interrupt *will* unmask the SYSTEM interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller functions

The source file `src/sam9_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_irq_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function, where the `level` and `up` arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge

level	up	interrupt on
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded, and as a result, the `hal_interrupt_acknowledge` function is empty.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Using the Advanced Interrupt Controller for VSRs

The SAM9 HAL has been designed to exploit benefits of the on-chip Advanced Interrupt Controller (AIC) on the SAM9. Support has been included for exploiting its ability to provide hardware vectoring for VSR interrupt handlers.

This support is dependent on definitions that may only be provided by the platform HAL and therefore is only enabled if the platform HAL package implements the `CYGINT_HAL_SAM9_AIC_VSR` CDL interface. The necessary definitions are available to all platform HALs which use the facilities of the [sam9_init.inc header file](#).

The interrupt decoding path has been optimised by allowing the AIC to be interrogated for the interrupt handler VSR to use. These vectored interrupts are by default still configured to point to the default ARM architecture HAL IRQ and FIQ VSRs. However applications may set their own VSRs to override this default behaviour to allow optimised interrupt handling.

The VSR vector numbers to use when overriding are defined as follows:

```
// FIQ is already defined as vector 7 in the architecture hal_intr.h
#define CYGNUM_HAL_VECTOR_SYSTEM 8 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_VECTOR_PIOA 9 // Parallel IO Controller A
#define CYGNUM_HAL_VECTOR_PIOB 10 // Parallel IO Controller B
#define CYGNUM_HAL_VECTOR_PIOC 11 // Parallel IO Controller C
// VSR 12 variant specific
#define CYGNUM_HAL_VECTOR_US0 13 // USART 0
#define CYGNUM_HAL_VECTOR_US1 14 // USART 1
#define CYGNUM_HAL_VECTOR_US2 15 // USART 2
#define CYGNUM_HAL_VECTOR_MCI 16 // Multimedia Card Interface
#define CYGNUM_HAL_VECTOR_UDP 17 // USB Device Port
#define CYGNUM_HAL_VECTOR_TWI 18 // Two-Wire Interface
#define CYGNUM_HAL_VECTOR_SPI 19 // Serial Peripheral Interface
#define CYGNUM_HAL_VECTOR_SPI1 20 // Serial Peripheral Interface
#define CYGNUM_HAL_VECTOR_SSC0 21 // Serial Synchronous Controller 0
// VSR 22 variant specific
// VSR 23 variant specific
#define CYGNUM_HAL_VECTOR_TC0 24 // Timer Counter 0
#define CYGNUM_HAL_VECTOR_TC1 25 // Timer Counter 1
#define CYGNUM_HAL_VECTOR_TC2 26 // Timer Counter 2
#define CYGNUM_HAL_VECTOR_UHP 27 // USB Host port
```

```

// VSRs 28..35 variant specific
#define CYGNUM_HAL_VECTOR_IRQ0 36 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_VECTOR_IRQ1 37 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_VECTOR_IRQ2 38 // Advanced Interrupt Controller (IRQ2)

// Variant specific vectors

#if defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9260)

#define CYGNUM_HAL_VECTOR_ADC 12 // Analog to Digital Converter
// Vector 22 unused
// Vector 23 unused

#define CYGNUM_HAL_VECTOR_EMAC 28 // Ethernet MAC
#define CYGNUM_HAL_VECTOR_ISI 29 // Image Sensor Interface
#define CYGNUM_HAL_VECTOR_US3 30 // USART 3
#define CYGNUM_HAL_VECTOR_US4 31 // USART 4
#define CYGNUM_HAL_VECTOR_US5 32 // USART 5
#define CYGNUM_HAL_VECTOR_TC3 33 // Timer Counter 3
#define CYGNUM_HAL_VECTOR_TC4 34 // Timer Counter 4
#define CYGNUM_HAL_VECTOR_TC5 35 // Timer Counter 5

#elif defined(CYGHWR_HAL_ARM_ARM9_SAM9_SAM9261)

// Vector 12 unused
#define CYGNUM_HAL_VECTOR_SSC1 22 // Serial Synchronous Controller 1
#define CYGNUM_HAL_VECTOR_SSC2 23 // Serial Synchronous Controller 2
#define CYGNUM_HAL_VECTOR_LCD 28 // LCD controller
// Vectors 29..35 unused

#else

#error Unknown SAM9 variant

#endif

```

Consult the kernel and generic HAL documentation for more information on VSRs and how to set them.

Interrupt handling withing standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers [above](#). Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the [interrupt vector number](#). The `data` argument is extracted from a corresponding table named

`hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Timers

Name

Timers — Use of on-chip timers

Periodic Interval Timer

The eCos kernel system clock is implemented using the Periodic Interval Timer (PIT). By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

The PIT is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using timer counter 1 (TC1). If the `gprof` package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then TC1 is reserved for use by the profiler.

Serial UARTs

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

Support is included in this processor HAL package for the SAM9's on-chip debug unit UART and up to four serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the SAM9 processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

This driver is contained in the `CYGPKG_IO_SERIAL_ARM_AT91` package. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Note that unlike the USART devices, the serial debug port does not support modem control signals such as those used for hardware signals. In addition, USART devices for a particular platform may also not have these control signals brought out to the physical serial port.

Multimedia Card Interface (MCI) driver

Name

Multimedia Card Interface (MCI) driver — Using MMC/SD cards with block drivers and filesystems

Overview

The MultiMedia Card Interface (MCI) driver in the SAM9 processor HAL allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

Configuration

This driver provides the necessary support for the generic MMC bus layer within the `CYGPKG_DEVS_DISK_MMC` package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package `CYGPKG_IO_DISK` is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are `"/dev/mmc0/1"` to refer to the first partition on the card, or `"/dev/mmc0/0"` to address the whole device including potentially the partition table at the start.

The driver may be forcibly disabled within this processor HAL package with the configuration option `CYGPKG_HAL_ARM_ARM9_SAM9_MCI`.

If the driver is enabled, there are only two SAM9 specific options:

`CYGIMP_HAL_ARM_ARM9_SAM9_MCI_INTMODE`

This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MCI driver in an initialisation context.

`CYGNUM_HAL_ARM_ARM9_SAM9_MCI_POWERSAVE_DIVIDER`

The SAM9 MCI peripheral allows the MCI clock to be divided down if told to enter power saving mode. This option specifies the divider to use. The driver itself does not implement any power saving - it is up to the application to enable power saving in the MCI control register if it is required.

Usage notes

MMC/SD cards may only be used in a MMC/SD card slot, and not a dataflash slot. The driver will detect the appropriate card sizes. Hotswapping of cards is supported by the driver, and in the case of eCosPro, the FAT

Multimedia Card Interface (MCI) driver

filesystem. Although any cards removed before explicit unmounting or a `sync()` call to flush filesystem buffers will likely result in a corrupted filesystem on the removed card.

The MMC/SD bus layer will parse partition tables, although it can be configured to allow handling of cards with no partition table.

Two-Wire Interface (TWI) driver

Name

Two-Wire Interface (TWI) driver — Configuration and implementation details of TWI (I²C®) driver

Overview

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the SAM9. This type of bus is also known as I²C®. The API for this may be found within the `CYGPKG_IO_I2C` package.

I²C®/TWI driver configuration

The I²C® driver uses the SAM9's internal Two-Wire Interface (TWI) support. This is controlled within the SAM9 processor HAL (`CYGPKG_HAL_SAM9`). The `CYGPKG_HAL_SAM9_TWI` CDL component controls whether the TWI driver is enabled. Within that component, there are two sub-options:

- `CYGNUM_HAL_SAM9_TWI_CLOCK` sets the speed of the TWI bus clock in Hz. This is usually 100kHz, but can be set up to 400kHz if the devices on the bus support this speed, also known as fast mode. However other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.
- The second option within the `CYGPKG_HAL_SAM9_TWI` component is `CYGNUM_HAL_SAM9_TWI_CKDIV`. This is the clock divider used when configuring the `TWI_CWGR` register. Consult the SAM9 datasheet description of the `TWI_CWGR` register for the formula used to determine the clock frequency. Increasing the divider will decrease the accuracy in practice of the generated I²C bus clock compared to `CYGNUM_HAL_SAM9_TWI_CLOCK`. But the divider must also be sufficiently low that the relevant factors do not overflow valid values for `CHDIV/CLDIV` in `TWI_CWGR`. Note that when the SAM9 is using a 60MHz MCK, then for 100kHz operation, a value for this option of 1 is most appropriate. For 400kHz, a value for this option of 0 is most appropriate. The default value of this CDL is an appropriate value for `CKDIV` assuming a 60MHz MCK and a TWI clock between 29kHz and 400kHz.

To be specific, the `CLDIV/CHDIV` fields of the `TWI_CWGR` are considered equal. The value of, for example, `CLDIV`, can be expressed as:

$$CLDIV = \frac{f_{MCK} - 6f_{TWI}}{2^{CKDIV+1} \cdot f_{TWI}}$$

To use the I²C/TWI driver, the generic I²C driver package `CYGPKG_IO_I2C` must be used. Documentation for its API may be found elsewhere.

Usage notes

Due to the characteristics of the SAM9's operation, it is not possible to provide support for repeated starts with the I²C package API. Similarly indicating a NACK when performing a receive is equivalent to also sending a STOP.

A test application for use with the AT24C512 serial EEPROM fitted to the AT91SAM9260EK board is provided within the `tests` subdirectory of the `CYGPKG_HAL_SAM9` package. This test communicates with the I²C EEPROM on the board to perform read and write operations using I²C. This test is not built by default. It may be built by enabling the configuration option `CYGBLD_HAL_ARM_ARM9_SAM9_TEST_TWI_AT24C512` within the SAM9 processor HAL.

Power saving support

Name

Power saving support — Extensions for saving power

Overview

There is support in the SAM9 processor HAL for a simple power saving mechanism. This is provided by two functions:

```
#include <cyg/hal/hal_intr.h>

__externC void cyg_hal_sam9_powersave_init( cyg_uint32 ip_addr );

__externC void cyg_hal_sam9_powerdown( void );
```

The powersaving system is initialized by calling `cyg_hal_sam9_powersave_init()`. The argument should be the IP address of this machine in network order. This can usually be fetched from the bootp data for an interface after completion of the call to `init_all_network_interfaces()`. e.g. `eth0_bootp_data.bp_ciaddr.s_addr`.

A call to `cyg_hal_sam9_powerdown()` will put the machine into a low power mode. This will involve switching to a slower system clock speed, disabling all peripherals except those that are defined to cause the system to wake up and return from this function.

Configuration

The exact behaviour of the power saving system is controlled by the following configuration options:

CYGPKG_HAL_ARM_ARM9_SAM9_POWERSAVE

This option controls the overall inclusion of the power saving system.

Default value: on

CYGSEM_HAL_ARM_ARM9_SAM9_POWERSAVE_POLL_ETHERNET

This option enables polling of the ethernet interface for relevant ARP packets and unicast IP packets. It is necessary for the CPU to run at a higher CPU speed for this option to work.

Default value: off

CYGSEM_HAL_ARM_ARM9_SAM9_POWERSAVE_IDLE

If this option is set, the CPU will go into idle mode, which will cause it to halt until an interrupt is delivered.

Default value: off

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_ACTIVE_DEVICES

This option defines the devices that are to be kept running during power down mode. An interrupt from one of these devices is usually the only way of bringing the system out of idle mode. The value of this option is a bit mask with bits set for each device that is to be kept active. The bits correspond to the peripheral identifiers described in the SAM9 documentation.

Default value: 0x00000000

CYGSEM_HAL_ARM_ARM9_SAM9_POWERSAVE_POLL_GPIO

This option control whether the power saving system will poll GPIO pins during power saving. For this to work the CPU cannot be put into idle mode.

Default value: on

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_PIO_HI

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 1, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_PIO_LO

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 0, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_SAM9_POWERSAVE_PIO_CHANGE

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to change between successive polls, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGBLD_HAL_ARM_ARM9_SAM9_TEST_POWERSAVE

This option controls whether a simple test is built to exercise power saving support. The test is not built by default as an external means is required to wake the processor up by one of the above configured mechanisms.

Default value: 0

XXVI. Atmel AT91SAM9260 Evaluation Kit Board Support

Overview

Name

eCos Support for the Atmel AT91SAM9260 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9260 Evaluation Kit. The AT91SAM9260 Evaluation Kit contains the AT91SAM9260 microprocessor, 64Mbytes of SDRAM, 256Mbytes of NAND flash memory, an Atmel Dataflash, an Atmel serial EEPROM, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for three serial channels (one debug, one full modem, one flow controlled), ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9260. eCos support for the many devices and peripherals on the boards and the AT91SAM9260 is described below.

For typical eCos development, a RedBoot image is programmed into the dataflash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The Dataflash consists of 8192 blocks of 1056 bytes each. In a typical setup, the first 33792 bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a 1056 block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J17 and DTE port at J20 (connected to USART channel 0) and flow controlled port at J18 (connected to USART channel 1) can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9260-EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9260-EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9260. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

The SAM9 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the AT91SAM9260-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the SAM9 processor HAL documentation.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (CYGPKG_DEVS_SPI_ARM_AT91) which in turn provides the underlying implementation for the SPI API layer in the CYGPKG_IO_SPI package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the CYGPKG_IO_FLASH package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9260-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM9260-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9260-EK board boots from the DataFlash and runs the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from Dataflash to SDRAM	redboot_ROM.ecm	redboot_ROM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is a historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9260 is only capable of loading programs from Dataflash or NAND flash into 4Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be **rebuilt** with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9260ek/dataflash/at91sam9260ek.h` within the `sam9260ek` HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/VERSION/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG

debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. Install it on a suitable PC running Windows XP.
2. Copy `dataflash_at91sam9260ek.bin` and `redboot_ROM.bin` to a suitable location on the Windows PC.
3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9260-EK board. Windows may ask you to install a new driver, in which case follow the instructions.
4. Start SAM-BA. Select "\\usb\\ARM0" for the communication interface, and "AT91SAM9260-EK" for the board. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".
5. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPIO CS1)", to program the on-board Dataflash device. Click Execute and SAM-BA should emit the following in the message area:

```
(AT91-ISP v1.13) 1 % DATAFLASH::Init 1
-I- DATAFLASH::Init 1 (trace level : 4)
-I- Loading applet isp-dataflash-at91sam9260.bin at address 0x20000000
-I- Memory Size : 0x840000 bytes
-I- Buffer address : 0x20002A70
-I- Buffer size: 0x80E80 bytes
-I- Applet initialization done
```

The actual options and output of SAM-BA may vary according to the version you are using. The behaviour documented here is that of SAM-BA 2.9.

6. Now select the DataFlash tab again, "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `dataflash_at91sam9260ek.bin` file and click "Open". The following output should be seen:

```
(AT91-ISP v1.13) 1 % GENERIC::SendBootFileGUI
GENERIC::SendFile dataflash_at91sam9260ek.bin at address 0x0
-I- File size : 0xF82 byte(s)
-I- Writing: 0xF82 bytes at 0x0 (buffer addr : 0x20002A70)
-I- 0xF82 bytes written by applet
```

7. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.
8. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.
9. In the Address field set the value to 0x8400.
10. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:


```
(AT91-ISP v1.13) 1 % send_file {DataFlash AT45DB/DCB} "redboot_ROM.bin" 0x8400 0
-I- Send File //bert/Shared/Releng/sam9260ek/redboot_ROM.bin at address 0x8400
GENERIC::SendFile //bert/Shared/Releng/sam9260ek/redboot_ROM.bin at address 0x8400
-I- File size : 0x243D0 byte(s)
-I- Writing: 0x243D0 bytes at 0x8400 (buffer addr : 0x20002A70)
-I- 0x243D0 bytes written by applet
```

11. Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

```
RomBOOT
>Start AT91Bootstrap...
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
No space to add 'net_device'
AT91_ETH: Waiting for PHY to reset.
AT91_ETH: Waiting for link to come up..
Ethernet eth0: MAC address 12:34:56:78:9a:bc
No IP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:50:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9260-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035d08-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x4083fbe0-0x4083ffff: .
... Program from 0x23ffffbe0-0x24000000 to 0x4083fbe0: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.83
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.11
Console baud rate: 115200
DNS domain name: farm.ecoscentric.com
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x0E:0x00:0x00:0xEA:0x18:0xF0
GDB connection port: 9000
Force console for special debug messages: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbef to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
Ethernet eth0: MAC address 0e:00:00:ea:18:a2
IP: 192.168.7.83/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>
```

```
RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:50:09, Sep 24 2009
```

```
Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.
```

```
Platform: AT91SAM9260-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035d08-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9260-EK are:

```
$ mkdir redboot_at91sam9260ek_rom
$ cd redboot_at91sam9260ek_rom
$ ecosconfig new at91sam9260ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the `sam9260ek` package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9260-EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/VERSION/AT91Bootstrap/board/at91sam9260ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9260ek.bin` which can be copied out.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9260-EK platform HAL package is loaded automatically when eCos is configured for the `sam9260ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-eabi-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM9260-EK board contains an 8Mbyte Atmel AT45DB DataFlash device. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9260-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9260-EK board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal RTTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9260-EK board uses the AT91SAM9260's internal USART serial support as described in the SAM9 processor HAL documentation. Three serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver; USART 0 which is mapped to virtual vector channel 1 and `"/dev/ser0"`; and USART 1 which is mapped to virtual vector channel 2 and `"/dev/ser1"`. Only USART 0 supports full modem control signals but USART 1 supports RTS/CTS.

MCI Driver

As the SAM9 MCI driver is part of the SAM9 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

Various options can be used to control specific of the SAM9 MCI driver. Consult the SAM9 HAL documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9260.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9260ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9260ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9260ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the **'reset'**, or by pressing the reset button) and the **'go'** command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

An issue occurs when the AT91 Ethernet driver is included in your configuration. In order to work around a board hardware design issue, the CPU generates an external reset in order to reset the Ethernet PHY. However this can be interpreted by the PEEDI as an indication that the CPU itself has reset, and if the PEEDI configuration file option `CORE0_STARTUP_MODE` is set to `RESET` then the CPU will be halted at this point. To avoid this issue, the `CORE0_STARTUP_MODE` can be set to `RUN`.

Consult the PEEDI documentation for information on other features.

Running ROM applications

Applications configured for ROM startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USARTs 0 or 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1 or 2.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9260-EK hardware, and should be read in conjunction with that specification. The AT91SAM9260-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the on-board flash and 0x50000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, MCI, and SPI dataflash facilities on the AT91SAM9260-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9260, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9260-EK. An on-board device and an external card slot are provided on the board. The on-chip device is typically used to contain RedBoot and flash configuration data. The external slot is available for application use.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 or 0x50000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    420 size  3920
Startup : Interrupt stack used    528 size  4096
Startup : Idlethread stack used    80 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 6.28 microseconds (38 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            64
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
5.07	3.38	8.70	0.87	46%	29%	Create thread
0.86	0.81	2.26	0.08	81%	81%	Yield thread [all suspended]
1.06	0.97	2.58	0.11	95%	56%	Suspend [suspended] thread
1.05	0.97	2.26	0.11	92%	67%	Resume thread
1.47	1.29	3.87	0.12	62%	25%	Set priority
0.41	0.16	0.81	0.10	92%	1%	Get priority
3.18	2.90	9.02	0.25	64%	62%	Kill [suspended] thread
0.85	0.81	1.93	0.08	79%	79%	Yield [no other] thread
1.65	1.45	3.54	0.13	73%	23%	Resume [suspended low prio] thread
1.06	0.97	2.42	0.11	95%	60%	Resume [runnable low prio] thread
1.45	1.29	3.70	0.09	64%	28%	Suspend [runnable] thread
0.87	0.81	2.09	0.09	73%	73%	Yield [only low prio] thread
1.06	0.97	2.42	0.11	92%	60%	Suspend [runnable->not runnable]
3.09	2.74	9.18	0.26	84%	54%	Kill [runnable] thread
2.42	2.09	5.80	0.15	39%	1%	Destroy [dead] thread
4.27	3.87	10.47	0.29	84%	42%	Destroy [runnable] thread
6.09	5.48	11.44	0.35	75%	45%	Resume [high priority] thread
2.00	1.93	4.51	0.09	73%	73%	Thread switch
0.16	0.16	0.64	0.01	99%	99%	Scheduler lock
0.68	0.64	0.97	0.05	80%	80%	Scheduler unlock [0 threads]
0.68	0.64	0.97	0.05	80%	80%	Scheduler unlock [1 suspended]

0.68	0.64	1.13	0.05	80%	80%	Scheduler unlock [many suspended]
0.68	0.64	0.97	0.05	80%	80%	Scheduler unlock [many low prio]
0.29	0.16	1.13	0.09	62%	34%	Init mutex
1.05	0.81	2.09	0.13	84%	9%	Lock [unlocked] mutex
1.20	0.97	2.90	0.15	81%	15%	Unlock [locked] mutex
1.00	0.81	2.42	0.12	62%	21%	Trylock [unlocked] mutex
0.93	0.81	1.93	0.11	53%	43%	Trylock [locked] mutex
0.20	0.16	0.64	0.07	81%	81%	Destroy mutex
5.72	5.32	7.73	0.21	68%	21%	Unlock/Lock mutex
0.43	0.32	1.61	0.12	96%	53%	Create mbox
0.33	0.16	0.64	0.07	65%	18%	Peek [empty] mbox
1.26	1.13	2.42	0.16	81%	81%	Put [first] mbox
0.32	0.16	0.64	0.06	65%	18%	Peek [1 msg] mbox
1.27	1.13	2.74	0.13	43%	46%	Put [second] mbox
0.32	0.16	0.64	0.07	65%	21%	Peek [2 msgs] mbox
1.32	1.13	3.22	0.16	62%	34%	Get [first] mbox
1.33	1.13	3.22	0.15	75%	21%	Get [second] mbox
1.11	0.97	2.09	0.12	43%	40%	Tryput [first] mbox
1.12	0.97	2.09	0.13	37%	43%	Peek item [non-empty] mbox
1.21	0.97	2.42	0.14	84%	6%	Tryget [non-empty] mbox
0.99	0.81	1.77	0.08	71%	12%	Peek item [empty] mbox
1.04	0.97	2.26	0.11	96%	75%	Tryget [empty] mbox
0.33	0.16	0.81	0.10	56%	25%	Waiting to get mbox
0.35	0.16	0.64	0.07	71%	9%	Waiting to put mbox
0.60	0.48	1.93	0.13	93%	56%	Delete mbox
4.33	3.87	8.70	0.37	84%	71%	Put/Get mbox
0.31	0.16	1.29	0.08	68%	28%	Init semaphore
0.87	0.81	2.09	0.10	96%	84%	Post [0] semaphore
0.95	0.81	2.09	0.11	56%	37%	Wait [1] semaphore
0.85	0.64	2.09	0.10	78%	9%	Trywait [0] semaphore
0.82	0.64	1.29	0.03	93%	3%	Trywait [1] semaphore
0.31	0.16	0.97	0.08	65%	25%	Peek semaphore
0.22	0.16	0.81	0.09	75%	75%	Destroy semaphore
3.43	3.06	5.96	0.22	87%	37%	Post/Wait semaphore
0.42	0.32	1.61	0.12	96%	62%	Create counter
0.29	0.16	0.97	0.09	62%	34%	Get counter value
0.24	0.16	0.81	0.11	90%	65%	Set counter value
1.06	0.81	2.09	0.13	84%	9%	Tick counter
0.30	0.16	1.45	0.11	56%	37%	Delete counter
0.28	0.16	1.29	0.11	50%	46%	Init flag
0.96	0.81	2.74	0.13	50%	43%	Destroy flag
0.82	0.64	1.93	0.12	53%	28%	Mask bits in flag
0.97	0.81	2.09	0.10	56%	31%	Set bits in flag [no waiters]
1.28	1.13	2.74	0.10	62%	34%	Wait for flag [AND]
1.28	1.13	2.90	0.11	62%	34%	Wait for flag [OR]
1.28	1.13	2.90	0.12	56%	37%	Wait for flag [AND/CLR]
1.26	1.13	2.90	0.13	96%	50%	Wait for flag [OR/CLR]
0.17	0.16	0.32	0.01	96%	96%	Peek on flag
0.60	0.48	1.93	0.13	93%	53%	Create alarm

```

1.70    1.45    4.35    0.20    75%   71% Initialize alarm
0.98    0.81    2.42    0.10    68%   25% Disable alarm
1.65    1.45    4.99    0.22    96%   90% Enable alarm
1.16    0.97    2.74    0.13    71%   21% Delete alarm
1.07    0.97    1.45    0.09    56%   40% Tick counter [1 alarm]
4.89    4.83    5.96    0.10    81%   81% Tick counter [many alarms]
1.89    1.77    3.06    0.13    93%   56% Tick & fire counter [1 alarm]
29.89   29.80   30.93    0.11    96%   62% Tick & fire counters [>1 together]
5.70    5.64    7.25    0.11    96%   87% Tick & fire counters [>1 separately]
5.66    5.64    7.73    0.04    97%   97% Alarm latency [0 threads]
6.43    5.80    8.54    0.37    54%   37% Alarm latency [2 threads]
14.05   12.56   15.79    0.81    46%   33% Alarm latency [many threads]
8.96    8.86    15.14    0.15    96%   78% Alarm -> thread resume latency

2.26    1.45    6.12    0.00                Clock/interrupt latency

2.69    1.77    6.93    0.00                Clock DSR latency

33      0      292 (main stack: 1388) Thread stack used (8016 total)
All done, main stack : stack used 1388 size 3920
All done : Interrupt stack used 208 size 4096
All done : Idlethread stack used 268 size 2048

```

Timing complete - 30140 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM9260-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXVII. Atmel AT91SAM9261 Evaluation Kit Board Support

Overview

Name

eCos Support for the Atmel AT91SAM9261 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9261 Evaluation Kit. The AT91SAM9261 Evaluation Kit contains the AT91SAM9261 microprocessor, 64Mbytes of SDRAM, 256Mbytes of NAND flash memory, an Atmel Dataflash, a Davicom DM9000 MAC+PHY, a SD/MMC/DataFlash socket, a DAC, an LCD display, external connections for a DEBUG serial channel, ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9261. eCos support for the many devices and peripherals on the boards and the AT91SAM9261 is described below.

For typical eCos development, a RedBoot image is programmed into the on-board dataflash memory, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The on-board Dataflash consists of 8192 blocks of 1056 bytes each. In a typical setup, the first 32K bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a 1056 block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J15 can be used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9261EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_DAVICOM_DM9000` for the DM9000 ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9261EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9261. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

The SAM9 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the SAM9261-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the SAM9 processor HAL documentation.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (CYGPKG_DEVS_SPI_ARM_AT91) which in turn provides the underlying implementation for the SPI API layer in the CYGPKG_IO_SPI package. All these packages are automatically loaded when configuring for the board..

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the CYGPKG_IO_FLASH package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

In general, devices (Caches, PIO, UARTs) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9261-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM9261-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9261-EK board boots from the DataFlash and run the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from Dataflash to SDRAM	redboot_ROM.ecm	redboot_ROM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is an historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9261 is only capable of loading programs from Dataflash or NAND flash into on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be rebuilt with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9261ek/dataflash/at91sam9261ek.h` within the `sam9260ek` HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/VERSION/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG

debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. Install it on a suitable PC running Windows XP.
2. Copy `dataflash_at91sam9261ek.bin` and `redboot_ROM.bin` to a suitable location on the Windows PC.
3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9261-EK board.
4. Power up or reset the board and Windows should recognize the USB device. If it does not, then move J21 to the 2-3 position and reset the board, it should be recognized now. Windows may ask you to install a new driver, in which case follow the instructions.
5. Start SAM-BA. Select "\\usb\\ARM0" for the communication interface, and "AT91SAM9261-EK" for the board. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".
6. If you moved J21 to 2-3, move it back to the default 1-2 position.
7. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPI0 CS0)", to program the on-board Dataflash device. Click Execute and SAM-BA should emit the following in the message area:

```
(AT91-ISP v1.13) 1 % DATAFLASH::Init 0
-I- DATAFLASH::Init 0 (trace level : 4)
-I- Loading applet isp-dataflash-at91sam9261.bin at address 0x20000000
-I- Memory Size : 0x840000 bytes
-I- Buffer address : 0x20002A40
-I- Buffer size: 0x80E80 bytes
-I- Applet initialization done
```

8. Select "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the `dataflash_at91sam9261ek.bin` file and click "Open". The following output should be seen:

```
(AT91-ISP v1.13) 1 % GENERIC::SendBootFileGUI
GENERIC::SendFile dataflash_at91sam9261ek.bin at address 0x0
-I- File size : 0x10A2 byte(s)
-I- Writing: 0x10A2 bytes at 0x0 (buffer addr : 0x20002A40)
-I- 0x10A2 bytes written by applet
```

9. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.
10. In the "Send File Name" box type in the path name to the `redboot_ROM.bin` file, or use the Open Folder button and browse to it.
11. In the Address field set the value to 0x8400.

- Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:

```
(AT91-ISP v1.13) 1 % send_file {DataFlash AT45DB/DCB} "redboot_ROM.bin" 0x8400 0
-I- Send File //bert/Shared/Releng/sam9261ek/redboot_ROM.bin at address 0x8400
GENERIC::SendFile //bert/Shared/Releng/sam9261ek/redboot_ROM.bin at address 0x8400
-I- File size : 0x1F928 byte(s)
-I- Writing: 0x1F928 bytes at 0x8400 (buffer addr : 0x20002A40)
-I- 0x1F928 bytes written by applet
```

- Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

```
RomBOOT
>Start AT91Bootstrap...
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
>Start AT91Bootstrap...
No network interfaces found

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:53:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: SAM9261-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20031230-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

- Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x4083fbe0-0x4083ffff: .
... Program from 0x23ffffbe0-0x24000000 to 0x4083fbe0: .
RedBoot> fis list
```

Name	FLASH addr	Mem addr	Length	Entry point
(reserved)	0x40000000	0x40000000	0x00008000	0x00000000
RedBoot	0x40008000	0x40008000	0x00028380	0x00000000
RedBoot config	0x4083F7C0	0x4083F7C0	0x00000420	0x00000000
FIS directory	0x4083FBE0	0x4083FBE0	0x00000420	0x00000000

```
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
GDB connection port: 9000
Force console for special debug messages: false
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbef to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
RomBOOT
>Start AT91Bootstrap...
+Ethernet eth0: MAC address 00:03:47:df:32:a8
IP: 192.168.7.85/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:53:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.
```



```
Platform: SAM9261-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20031230-0x23ffef80 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9261-EK are:

```
$ mkdir redboot_at91sam9261ek_rom
$ cd redboot_at91sam9261ek_rom
$ ecosconfig new at91sam9261ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9261ek/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the AT91Bootstrap directory of the sam9260ek package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9261EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/VERSION/AT91Bootstrap/board/at91sam9261ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9261ek.bin` which can be copied out.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9261-EK platform HAL package is loaded automatically when eCos is configured for the `sam9261ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-eabi-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91SAM9261-EK board contains an 8Mbyte Atmel AT45DB DataFlash device. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9261-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9261-EK board uses a Davicom DM9000 ethernet MAC and PHY chip for ethernet connectivity. The `CYGPKG_DEVS_ETH_DAVICOM_DM9000` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9261-EK board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9261-EK board uses the AT91SAM9261's internal RTTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91SAM9261-EK board uses the AT91SAM9261's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9261-EK board uses the AT91SAM9261's internal USART serial support as described in the SAM9 processor HAL documentation. Just one serial port is available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver. This serial port only provides basic TX and RX lines, with no modem control or flow control lines. RTS/CTS.

MCI Driver

As the SAM9 MCI driver is part of the SAM9 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

Various options can be used to control specific of the SAM9 MCI driver. Consult the SAM9 HAL documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9261.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9261ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9261ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9261ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the **'reset'**, or by pressing the reset button) and the **'go'** command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

Consult the PEEDI documentation for information on other features.

Running ROM applications

Applications configured for ROM startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9261-EK hardware, and should be read in conjunction with that specification. The AT91SAM9261-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the on-board flash and 0x50000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

DM9000 Ethernet Device

This is located on chip select 2 of the static memory controller and is visible at physical address 0x30000000. The HAL uses the MMU to relocate this to 0x80000000 in the virtual memory space.

Off-chip Peripherals

eCos uses the SDRAM, MCI, and SPI dataflash facilities on the AT91SAM9261-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9261, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9261. An on-board device and an external card slot are provided on the board. The on-chip device is typically used to contain RedBoot and flash configuration data. The external slot is available for application use.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 or 0x50000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

RedBoot or applications can also be booted from the Dataflash card socket, as well as from the on-board Dataflash device. Booting from the Dataflash card socket can be performed by switching jumper J21 from pins 1-2 closed, to pins 2-3 closed. When booting RedBoot or applications in this way, you *must* enable the SAM9261-EK platform HAL configuration option "SPI chip select #0 is socket" (CYGHWR_HAL_ARM_ARM9_SAM9261EK_DATAFLASH_NPCS0_SOCKET) found within the "External Atmel AT49xxx DataFlash memory support" component. Failure to do so will not just render the Dataflash card inaccessible after booting, but is likely to cause permanent damage to the AT91SAM9261.

Once an appropriately configured "ROM" startup image has been built, it can be converted to raw binary format using **arm-eabi-objcopy**. You must then copy an AT91Bootstrap second stage boot loader binary image to the beginning (offset 0) of the card, and then the RedBoot/application image at offset 0x8000 on the card.

Programming on the card can be performed with SAM-BA, an external programmer, or with a standard build of RedBoot booted from the on-board Dataflash. An example of using an installed RedBoot loaded from on-board Dataflash to program the card would be as follows. Note the two Flash memories detected by RedBoot, as shown in the **version** output. The first is the on-board Dataflash part, the second is the Dataflash located in the card socket.

```
RedBoot> version
```

```
RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version UNKNOWN - built 15:10:18, Nov  2 2007
```

```
Platform: SAM9261-EK (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007 eCosCentric Limited
```

```
RAM: 0x20000000-0x24000000, [0x200305d0-0x23ffef80] available
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
FLASH: 0x50000000-0x5083ffff, 8192 x 0x420 blocks
```

```
RedBoot> fis list
```

Name	FLASH addr	Mem addr	Length	Entry point
(reserved)	0x40000000	0x40000000	0x00008000	0x00000000
RedBoot	0x40008000	0x20030800	0x00028380	0x20030800
RedBoot config	0x4083F7C0	0x4083F7C0	0x00000420	0x00000000
FIS directory	0x4083FBE0	0x4083FBE0	0x00000420	0x00000000

```
RedBoot> fis load -b ${freememlo} (reserved)
```

```
RedBoot> fis write -b ${freememlo} -f 0x50000000 -l 0x8000
```

```
* CAUTION * about to program FLASH
```

```
      at 0x50000000..0x500083ff from 0x20030800 - continue (y/n)? y
```

```
... Erase from 0x50000000-0x500083ff: .....
```

```
... Program from 0x20030800-0x20038c00 to 0x50000000: .....
```

```
RedBoot> load -r -m tftp -b ${freememlo} /sam9261ek/redboot.bin
```

```
Raw file loaded 0x20030800-0x2004f3a3, assumed entry at 0x20030800
```

```
RedBoot> fis write -b ${freememlo} -f 0x50008000 -l 0x28380
```

```
* CAUTION * about to program FLASH
```

```
      at 0x50008000..0x5003037f from 0x20030800 - continue (y/n)? y
```

```
... Erase from 0x50007fe0-0x5003037f: .....
```

```
... Program from 0x20030800-0x20058b80 to 0x50008000: .....
```

```
RedBoot>
```

After this, the board can be powered off, jumper J21 switched to pins 2-3, and the board powered up again. The application or RedBoot will then boot from the Dataflash card.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    420 size  3920
Startup : Interrupt stack used    536 size  4096
Startup : Idlethread stack used     80 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

```
Reading the hardware clock takes 1 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took      6.37 microseconds (39 raw clock ticks)
```

Testing parameters:

```
Clock samples:          32
Threads:                64
Thread switches:        128
Mutexes:                32
Mailboxes:              32
Semaphores:             32
Scheduler operations:    128
Counters:               32
Flags:                  32
Alarms:                 32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
4.95	3.38	6.44	0.81	48%	26%	Create thread
0.84	0.81	1.93	0.06	85%	85%	Yield thread [all suspended]
1.04	0.97	2.26	0.10	98%	64%	Suspend [suspended] thread
0.95	0.81	1.61	0.07	65%	23%	Resume thread
1.35	1.13	3.87	0.12	92%	6%	Set priority
0.26	0.16	0.64	0.10	92%	50%	Get priority
2.98	2.58	8.05	0.24	87%	39%	Kill [suspended] thread
0.85	0.81	1.61	0.07	79%	79%	Yield [no other] thread
1.48	1.29	2.42	0.09	64%	12%	Resume [suspended low prio] thread
0.93	0.81	1.13	0.07	65%	29%	Resume [runnable low prio] thread
1.17	0.97	1.93	0.09	62%	10%	Suspend [runnable] thread
0.84	0.81	1.61	0.05	87%	87%	Yield [only low prio] thread
0.88	0.81	1.29	0.09	93%	59%	Suspend [runnable->not runnable]
2.85	2.58	6.60	0.18	81%	68%	Kill [runnable] thread

2.16	1.93	4.67	0.15	75%	17%	Destroy [dead] thread
3.92	3.70	6.77	0.16	68%	70%	Destroy [runnable] thread
5.76	5.32	10.15	0.29	81%	32%	Resume [high priority] thread
2.02	1.93	4.19	0.10	98%	54%	Thread switch
0.21	0.16	0.64	0.07	74%	74%	Scheduler lock
0.69	0.64	1.29	0.07	75%	75%	Scheduler unlock [0 threads]
0.69	0.64	0.97	0.06	74%	74%	Scheduler unlock [1 suspended]
0.65	0.64	0.81	0.00	99%	99%	Scheduler unlock [many suspended]
0.68	0.64	0.97	0.05	79%	79%	Scheduler unlock [many low prio]
0.31	0.16	1.29	0.08	71%	25%	Init mutex
1.03	0.81	2.58	0.14	84%	12%	Lock [unlocked] mutex
1.13	0.97	2.58	0.09	68%	28%	Unlock [locked] mutex
0.92	0.81	1.93	0.11	50%	46%	Trylock [unlocked] mutex
0.82	0.81	0.97	0.02	93%	93%	Trylock [locked] mutex
0.20	0.16	0.64	0.07	81%	81%	Destroy mutex
5.06	4.99	6.44	0.12	93%	90%	Unlock/Lock mutex
0.42	0.32	1.61	0.12	96%	62%	Create mbox
0.33	0.16	1.13	0.11	50%	28%	Peek [empty] mbox
1.08	0.97	2.26	0.12	93%	53%	Put [first] mbox
0.19	0.16	0.64	0.05	90%	90%	Peek [1 msg] mbox
1.03	0.97	1.93	0.10	96%	75%	Put [second] mbox
0.18	0.16	0.32	0.04	87%	87%	Peek [2 msgs] mbox
1.08	0.97	2.26	0.12	93%	56%	Get [first] mbox
1.03	0.97	1.45	0.09	65%	65%	Get [second] mbox
0.89	0.81	1.61	0.10	96%	59%	Tryput [first] mbox
0.88	0.81	1.93	0.11	96%	71%	Peek item [non-empty] mbox
0.97	0.81	1.77	0.06	78%	15%	Tryget [non-empty] mbox
0.85	0.81	1.13	0.07	75%	75%	Peek item [empty] mbox
0.91	0.81	1.77	0.11	96%	53%	Tryget [empty] mbox
0.19	0.16	0.32	0.04	84%	84%	Waiting to get mbox
0.19	0.16	0.48	0.05	84%	84%	Waiting to put mbox
0.41	0.32	1.29	0.11	96%	62%	Delete mbox
3.59	3.38	6.60	0.19	65%	96%	Put/Get mbox
0.26	0.16	0.81	0.09	50%	46%	Init semaphore
0.84	0.64	1.77	0.07	84%	6%	Post [0] semaphore
0.92	0.81	1.93	0.11	50%	46%	Wait [1] semaphore
0.82	0.64	1.45	0.05	87%	6%	Trywait [0] semaphore
0.82	0.81	1.29	0.03	96%	96%	Trywait [1] semaphore
0.28	0.16	0.81	0.09	59%	37%	Peek semaphore
0.20	0.16	0.48	0.06	78%	78%	Destroy semaphore
3.29	3.06	5.15	0.13	90%	3%	Post/Wait semaphore
0.43	0.32	1.61	0.12	96%	53%	Create counter
0.29	0.16	0.64	0.11	40%	43%	Get counter value
0.20	0.16	0.64	0.07	81%	81%	Set counter value
0.97	0.81	1.61	0.05	81%	12%	Tick counter
0.20	0.16	0.48	0.06	81%	81%	Delete counter
0.26	0.16	0.97	0.10	96%	50%	Init flag
0.91	0.81	2.09	0.12	96%	59%	Destroy flag
0.76	0.64	1.29	0.09	59%	37%	Mask bits in flag

```

0.91    0.81    1.45    0.10    46%    46% Set bits in flag [no waiters]
1.28    1.13    2.58    0.09    65%    31% Wait for flag [AND]
1.23    1.13    1.61    0.09    46%    46% Wait for flag [OR]
1.26    1.13    1.77    0.08    65%    28% Wait for flag [AND/CLR]
1.24    1.13    1.77    0.09    56%    40% Wait for flag [OR/CLR]
0.17    0.16    0.32    0.02    93%    93% Peek on flag

0.60    0.48    2.09    0.13    96%    53% Create alarm
1.67    1.45    3.87    0.25    84%    65% Initialize alarm
0.87    0.81    1.93    0.10    81%    81% Disable alarm
1.41    1.29    3.22    0.15    93%    59% Enable alarm
0.97    0.81    1.93    0.07    75%    18% Delete alarm
1.09    0.97    1.61    0.08    62%    34% Tick counter [1 alarm]
4.89    4.83    5.32    0.08    71%    71% Tick counter [many alarms]
1.87    1.77    3.06    0.12    96%    62% Tick & fire counter [1 alarm]
29.72   29.64   30.44    0.10    96%    59% Tick & fire counters [>1 together]
5.67    5.64    6.28    0.05    90%    90% Tick & fire counters [>1 separately]
5.81    5.80    7.25    0.02    98%    98% Alarm latency [0 threads]
5.87    5.80    7.89    0.13    91%    88% Alarm latency [2 threads]
9.28    8.22    11.28    0.58    49%    30% Alarm latency [many threads]
8.91    8.86    13.05    0.09    96%    96% Alarm -> thread resume latency

1.66    1.45    3.54    0.00                    Clock/interrupt latency

2.40    1.93    5.15    0.00                    Clock DSR latency

33      0      312 (main stack: 1416) Thread stack used (8016 total)
All done, main stack : stack used 1416 size 3920
All done : Interrupt stack used 208 size 4096
All done : Idlethread stack used 804 size 2048

```

Timing complete - 30040 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM9261-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXVIII. Atmel AT91SAM9263 Evaluation Kit Board Support

Overview

Name

eCos Support for the Atmel AT91SAM9263 Evaluation Kit — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91SAM9263 Evaluation Kit. The AT91SAM9263 Evaluation Kit contains the AT91SAM9263 microprocessor, 64Mbytes of SDRAM, 4Mbytes of PSRAM, 256Mbytes of NAND flash memory, an Atmel serial EEPROM, a Davicom DM9161A PHY, one SD/MMC/DataFlash socket and a SD/MMC socket, a DAC, external connections for two serial channels (one debug channel and one flow controlled), ethernet, USB host/device, and the various other peripherals supported by the AT91SAM9263. eCos support for the many devices and peripherals on the boards and the AT91SAM9263 is described below.

For typical eCos development, a RedBoot image is programmed onto a dataflash card in the SD/MMC/DataFlash socket, and the board will load this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the SAM9 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

Bootimg is from a DataFlash card in the SD/MMC/DataFlash socket. In a typical setup, the first 32K bytes are reserved for the second-level bootstrap, AT91Bootstrap. The following 164736 bytes are reserved for the use of the ROM RedBoot image (The odd size aligns the end of the RedBoot area to a block boundary). The topmost block is used to manage the flash and the next block down holds RedBoot **fconfig** values. The remaining blocks can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J14 and flow controlled port at J18 (connected to USART channel 0) can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91SAM9263EK target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the AT91SAM9263EK board.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC`. This driver is also loaded automatically when configuring for the board.

There is a driver for the on-chip real-time timer controller (RTTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC`. This driver is also loaded automatically when configuring for the target.

The SAM9 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91SAM9263. This type of bus is also known as I²C®. Further documentation may be found in the SAM9 processor HAL documentation.

The SAM9 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the SAM9263-EK target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the SAM9 processor HAL documentation. The driver can be configured to use either the SD/MMC/DataFlash socket at J19 or the SD/MMC socket at J10. By default it uses J10, leaving J9 for the bootstrap Dataflash.

The platform HAL provides definitions to allow access to devices on the SPI bus. The HAL provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the board.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package as well as the above SPI packages. That package is automatically loaded when configuring for the target. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the `CYGPKG_IO_FLASH` package, if that package is loaded in the configuration.

It is also possible to configure the HAL to access MMC cards in SPI mode, instead of using the MCI interface.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91SAM9263-EK support is intended to work with GNU tools configured for an arm-eabi target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91SAM9263-EK board for eCos Development

Overview

In a typical development environment, the AT91SAM9263-EK board boots from a 4MiB DataFlash card and run the RedBoot ROM monitor from SDRAM. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-eabi-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot loaded from 4MiB Dataflash to SDRAM	redboot_ROM.ecm	redboot_ROM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Note that the use of the term ROM for the initial RedBoot configuration is an historical accident. RedBoot actually runs from SDRAM after being loaded there from Dataflash by the second-level bootstrap. The use of ROM for this configuration is intended to indicate that it initializes the microprocessor and board peripherals, as opposed to the RAM configuration which assumes that this has already been done.

Initial Installation

The on-chip boot program on the AT91SAM9263 is only capable of loading programs from DataFlash, SD card or NAND flash into on-chip SRAM and is therefore quite restrictive. Consequently RedBoot cannot be booted directly and a second-level bootstrap must be used. Such a second-level bootstrap is supplied by Atmel in the form of AT91Bootstrap. This is therefore programmed into the start of Dataflash and is then responsible for initializing the SDRAM and loading RedBoot from Dataflash and executing it.

Caution

There is a size limit on the size of applications which the AT91Bootstrap second level bootstrap will load. Images larger than 320Kbytes will require the AT91Bootstrap application to be **rebuilt** with a larger `IMG_SIZE` definition in `AT91Bootstrap/board/at91sam9263ek/dataflash/at91sam9263ek.h` within the `sam9260ek` HAL package in the eCos source repository (`packages/hal/arm/arm9/sam9260ek/VERSION/`).

There are basically two ways to write the second-level bootstrap and RedBoot to the Dataflash. The first is to use the Atmel-supplied SAM-BA program that interacts with the on-chip boot program. The second is to use a JTAG debugger that understands the microcontroller and can write to the dataflash (for example the Ronetix PEEDI). Since the availability of the latter cannot be guaranteed, only the first method will be described here.

Programming RedBoot into DataFlash using SAM-BA

The following gives the steps needed to program the second-level bootstrap and RedBoot into the DataFlash card using SAM-BA. The user should refer to the SAM-BA documentation for full details of how to run the program.

1. Download the AT91 In-system Programmer software package from the Atmel website. Install it on a suitable PC running Windows XP.
2. Copy `dataflash_at91sam9263ek.bin` and `redboot_ROM.bin` to a suitable location on the Windows PC.
3. Connect a null-modem serial cable between the DEBUG serial port of the board and a serial port on a convenient host (which need not be the PC running SAM-BA). Run a terminal emulator (Hyperterm or minicom) at 115200 baud. Connect a USB cable between the PC and the AT91SAM9263-EK board. Windows may ask you to install a new driver, in which case follow the instructions.
4. Power up the board without the DataFlash card inserted. This will force the bootstrap to enter ISP mode.
5. Start SAM-BA. Select "`usb\ARM0`" for the communication interface, and "`AT91SAM9263-EK`" for the board. If the USB option does not appear, check the cable and look in the Windows Device Manager for the active device. If all is well, click on "Connect".
6. In the SAM-BA main window, select the "SDRAM" tab, select the "Enable SDRAM 100MHz" script from the dropdown menu and click Execute. SAM-BA should emit the following messages:

```
(AT91-ISP v1.10) 34 % SDRAM::initSDRAM_100
-I- Configure PIOD as peripheral (D16/D31)
-I- Init MATRIX to support EBI0 CS1 for SDRAM
-I- Init SDRAM
-I- 1. A minimum pause of 200us is provided to precede any signal toggle
-I- 2. A Precharge All command is issued to the SDRAM
-I- *pSDRAM = 0;
-I- 3. Eight Auto-refresh are provided
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- *pSDRAM = 0;
-I- 4. A mode register cycle is issued to program the SDRAM parameters
-I- *(pSDRAM+0x20) = 0;
-I- 5. Write refresh rate into SDRAMC refresh timer COUNT register
-I- 6. A Normal Mode Command is provided, 3 clocks after tMRD is set
-I- *pSDRAM = 0;
-I- End of Init_SDRAM_100
(AT91-ISP v1.10) 34 %
```

7. Now insert the DataFlash card into the socket at J9. In the SAM-BA main window, select the "DataFlash AT45DB/DBC" tab and in the "Scripts" dropdown menu select "Enable Dataflash (SPI0 CS0)", to program the Dataflash card. Click Execute and SAM-BA should emit the following in the message area:

```
(AT91-ISP v1.13) 1 % DATAFLASH::Init 0
-I- DATAFLASH::Init 0 (trace level : 4)
-I- Loading applet isp-dataflash-at91sam9263.bin at address 0x20000000
-I- Memory Size : 0x840000 bytes
-I- Buffer address : 0x20002A40
-I- Buffer size: 0x80E80 bytes
-I- Applet initialization done
```

The actual options and output of SAM-BA may vary according to the version you are using. The behaviour documented here is that of SAM-BA 2.9.

8. Select "Send BootFile" from the "Scripts" menu and "Execute" it. When the file open dialog appears, select the dataflash_at91sam9263ek.bin file and click "Open". The following output should be seen:

```
(AT91-ISP v1.13) 1 % GENERIC::SendBootFileGUI
GENERIC::SendFile dataflash_at91sam9263ek.bin at address 0x0
-I- File size : 0x106E byte(s)
-I- Writing: 0x106E bytes at 0x0 (buffer addr : 0x20002A40)
-I- 0x106E bytes written by applet
```

9. The second-level bootstrap has now been written to DataFlash, we must now write RedBoot.
10. In the "Send File Name" box type in the path name to the redboot_ROM.bin file, or use the Open Folder button and browse to it.
11. In the Address field set the value to 0x8400.
12. Click the "Send File" button. SAM-BA will put up a dialog box while it is writing the file to the DataFlash, and will output something similar to the following in the message area:

```
(AT91-ISP v1.13) 1 % send_file {DataFlash AT45DB/DCB} "redboot_ROM.bin" 0x8400 0
-I- Send File //bert/Shared/Releng/sam9263ek/redboot_ROM.bin at address 0x8400
GENERIC::SendFile //bert/Shared/Releng/sam9263ek/redboot_ROM.bin at address 0x8400
-I- File size : 0x24290 byte(s)
-I- Writing: 0x24290 bytes at 0x8400 (buffer addr : 0x20002A40)
-I- 0x24290 bytes written by applet
```

13. Shut down SAM-BA and disconnect the USB cable. Press the reset button on the board and something similar to the following should be output on the DEBUG serial line.

```
RomBOOT
>Start AT91Bootstrap...
+**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
No space to add 'net_device'
AT91_ETH: Waiting for PHY to reset.
AT91_ETH: Waiting for link to come up..
No network interfaces found

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:56:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
```

Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

```
Platform: AT91SAM9263-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035ba8-0x23fff5b0 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x4083fbe0-0x4083ffff: .
... Program from 0x23ffffbe0-0x24000000 to 0x4083fbe0: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS domain name: ecoscentric.com
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
Update RedBoot non-volatile configuration - continue (y/n)? y
... Erase from 0x4083f7c0-0x4083fbdf: .
... Program from 0x23fff7c0-0x23fffbfe0 to 0x4083f7c0: .
RedBoot>
```

The RedBoot installation is now complete. This can be tested by powering off the board, and then powering on the board again. Output similar to the following should be seen on the DEBUG serial port. Verify the IP settings are as expected.

```
RomBOOT
>Start AT91Bootstrap...
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 192.168.7.222/255.255.255.0, Gateway: 192.168.7.11
Default server: 192.168.7.11
DNS server IP: 192.168.7.11, DNS domain name: <null>

RedBoot(tm) bootstrap and debug environment [ROM]
eCosCentric certified release, version v3_0_3 - built 12:56:09, Sep 24 2009

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007, 2008 eCosCentric Limited
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: AT91SAM9263-EK (ARM9)
RAM: 0x20000000-0x24000000 [0x20035ba8-0x23fff5b0 available]
FLASH: 0x40000000-0x4083ffff, 8192 x 0x420 blocks
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROM version of RedBoot for the AT91SAM9263-EK are:

```
$ mkdir redboot_at91sam9263ek_rom
$ cd redboot_at91sam9263ek_rom
$ ecosconfig new at91sam9263ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/sam9263ek/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Rebuilding AT91Bootstrap

The sources of AT91Bootstrap are found in the `AT91Bootstrap` directory of the `sam9260ek` package. This is a copy of the software as supplied by Atmel with some slight modifications to permit it to be built with the same tools as eCos.

To rebuild the second-level bootstrap for the AT91SAM9263EK execute the following commands:

```
$ cd $ECOS_REPOSITORY/hal/arm/arm9/sam9260ek/VERSION/AT91Bootstrap/board/at91sam9263ek/dataflash
$ make
```

This should result in the creation of a number of files, including `dataflash_at91sam9263ek.bin` which can be copied out.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91SAM9263-EK platform HAL package is loaded automatically when eCos is configured for the `sam9263ek` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-eabi-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into DataFlash. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. This startup type can also be used for applications loaded via JTAG.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The board has an SD/MMC/DataFlash socket into which a dataflash card may be inserted. The `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91SAM9263-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91SAM9263-EK board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal RTTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91RTTC` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91WDTC` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91WDTC_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

Warning

The ATSAM926x processor will boot with watchdog support enabled, and the watchdog configuration is write-once. That is, if it is disabled, it cannot be re-enabled. Due to its nature, RedBoot disables the watchdog when it starts so any eCos applications with watchdog support enabled that are run by RedBoot will not function correctly.

USART Serial Driver

The AT91SAM9263-EK board uses the AT91SAM9263's internal USART serial support as described in the SAM9 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver; and USART 0 which is mapped to virtual vector channel 1 and `"/dev/ser0"`. The debug port is two wires only. but USART 0 supports RTS/CTS.

MCI Driver

As the SAM9 MCI driver is part of the SAM9 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

Various options can be used to control specific aspects of the SAM9 MCI driver. Consult the SAM9 HAL documentation for information on its configuration. The option CYGHWR_HAL_ARM_ARM9_SAM9_SAM9263_MCI controls which of the two MCI interfaces will be used by the driver.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`

The arm-eabi-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926EJ CPU in the AT91SAM9263.

`-mthumb`

The arm-eabi-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option CYGHWR_THUMB.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option CYGBLD_ARM_ENABLE_THUMB_INTERWORK.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded applications, including RedBoot. Only ROM configuration applications should be debugged using JTAG, RAM applications assume the presence of RedBoot.

Ronetix PEEDI notes

On the Ronetix PEEDI, the `peedi.at91sam9263ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `peedi.at91sam9263ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `CORE0_BREAKMODE` directive in the `[PLATFORM_ARM]` section. Edit this file if you wish to use hardware break points, and remember to restart the PEEDI to make the changes take effect.

On the PEEDI, debugging can be performed either via the telnet interface or using **arm-eabi-gdb** and the GDB interface. In the case of the latter, **arm-eabi-gdb** needs to connect to TCP port 2000 on the PEEDI's IP address. For example:

```
(gdb) target remote 111.222.333.444:2000
```

By default when the PEEDI is powered up, the target will always run the initialization section of the `peedi.at91sam9263ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset** command.

If the board is reset (either with the **'reset'**, or by pressing the reset button) and the **'go'** command is then given, then the board will boot as normal. If a second-level bootstrap and ROM RedBoot is resident in DataFlash, it will be run.

An issue occurs when the AT91 Ethernet driver is included in your configuration. In order to work around a board hardware design issue, the CPU generates an external reset in order to reset the Ethernet PHY. However this can be interpreted by the PEEDI as an indication that the CPU itself has reset, and if the PEEDI configuration file option `CORE0_STARTUP_MODE` is set to `RESET` then the CPU will be halted at this point. To avoid this issue, the `CORE0_STARTUP_MODE` can be set to `RUN`.

Consult the PEEDI documentation for information on other features.

Running ROM applications

Applications configured for ROM startup can be run directly under JTAG. Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USARTs 0 or 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1 or 2.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91SAM9263-EK hardware, and should be read in conjunction with that specification. The AT91SAM9263-EK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the SAM9 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x20040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is unused by eCos and is available for application use.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x40000000 for the dataflash slot, the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, ethernet PHY, MCI, and SPI dataflash facilities on the AT91SAM9263-EK board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91SAM9263, using the facilities of the SAM9 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91SAM9263. An external card slot are provided on the board which is typically used to contain RedBoot and flash configuration data.

Accesses to Dataflash are performed via the Flash API, using 0x40000000 as the nominal address of the device, although it does not truly exist in the processor address space. For the external card slot, on driver initialisation, eCos and RedBoot can detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x40000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    420 size  3920
Startup : Interrupt stack used    528 size  4096
Startup : Idlethread stack used    96 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 1 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 5.95 microseconds (37 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            64
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
4.89	3.20	7.84	0.85	48%	25%	Create thread
0.73	0.64	1.92	0.10	98%	54%	Yield thread [all suspended]
0.93	0.80	2.08	0.11	46%	42%	Suspend [suspended] thread
0.91	0.80	1.92	0.10	45%	46%	Resume thread
1.32	1.12	3.36	0.11	67%	14%	Set priority
0.27	0.16	0.64	0.10	42%	43%	Get priority
2.97	2.72	7.20	0.19	71%	65%	Kill [suspended] thread
0.71	0.64	0.96	0.08	56%	56%	Yield [no other] thread
1.51	1.28	3.20	0.12	87%	7%	Resume [suspended low prio] thread
0.89	0.80	1.44	0.09	95%	53%	Resume [runnable low prio] thread
1.27	1.12	2.24	0.08	60%	28%	Suspend [runnable] thread
0.72	0.64	1.44	0.09	98%	56%	Yield [only low prio] thread
0.93	0.80	1.60	0.09	57%	34%	Suspend [runnable->not runnable]
2.83	2.56	6.08	0.15	90%	4%	Kill [runnable] thread
2.27	2.08	5.44	0.14	70%	28%	Destroy [dead] thread
3.92	3.52	6.72	0.16	79%	15%	Destroy [runnable] thread
5.79	5.44	9.44	0.24	81%	48%	Resume [high priority] thread
1.88	1.76	3.68	0.09	61%	36%	Thread switch
0.08	0.00	0.32	0.08	50%	49%	Scheduler lock
0.56	0.48	1.12	0.08	99%	50%	Scheduler unlock [0 threads]
0.57	0.48	1.12	0.08	50%	49%	Scheduler unlock [1 suspended]

0.56	0.48	1.12	0.08	98%	55%	Scheduler unlock [many suspended]
0.57	0.48	1.28	0.08	99%	50%	Scheduler unlock [many low prio]
0.19	0.00	1.12	0.07	87%	6%	Init mutex
0.96	0.80	2.72	0.12	59%	37%	Lock [unlocked] mutex
1.08	0.80	2.88	0.15	87%	9%	Unlock [locked] mutex
0.86	0.64	1.92	0.12	84%	12%	Trylock [unlocked] mutex
0.76	0.64	1.28	0.08	62%	34%	Trylock [locked] mutex
0.08	0.00	0.32	0.08	96%	50%	Destroy mutex
4.95	4.80	7.68	0.23	90%	90%	Unlock/Lock mutex
0.28	0.16	1.28	0.11	53%	43%	Create mbox
0.19	0.00	0.48	0.07	71%	6%	Peek [empty] mbox
1.12	0.96	2.40	0.09	65%	28%	Put [first] mbox
0.19	0.00	0.48	0.05	81%	3%	Peek [1 msg] mbox
1.10	0.96	1.44	0.07	68%	25%	Put [second] mbox
0.21	0.00	0.64	0.08	68%	3%	Peek [2 msgs] mbox
1.17	0.96	2.56	0.12	81%	15%	Get [first] mbox
1.16	0.96	1.44	0.07	68%	6%	Get [second] mbox
0.97	0.80	1.92	0.08	71%	18%	Tryput [first] mbox
1.00	0.80	1.92	0.09	78%	6%	Peek item [non-empty] mbox
1.08	0.96	2.40	0.12	96%	50%	Tryget [non-empty] mbox
0.88	0.80	2.08	0.11	96%	71%	Peek item [empty] mbox
0.90	0.80	1.60	0.10	96%	53%	Tryget [empty] mbox
0.21	0.16	0.48	0.07	75%	75%	Waiting to get mbox
0.21	0.16	0.48	0.07	78%	78%	Waiting to put mbox
0.45	0.32	1.44	0.10	59%	37%	Delete mbox
3.50	3.36	6.56	0.20	96%	96%	Put/Get mbox
0.19	0.16	1.12	0.06	96%	96%	Init semaphore
0.73	0.64	1.76	0.12	93%	65%	Post [0] semaphore
0.81	0.64	1.76	0.08	71%	18%	Wait [1] semaphore
0.72	0.64	1.76	0.11	96%	71%	Trywait [0] semaphore
0.70	0.64	1.28	0.09	71%	71%	Trywait [1] semaphore
0.17	0.00	0.64	0.06	75%	12%	Peek semaphore
0.11	0.00	0.96	0.11	46%	46%	Destroy semaphore
3.15	3.04	5.60	0.18	93%	93%	Post/Wait semaphore
0.30	0.16	1.60	0.13	40%	46%	Create counter
0.16	0.00	0.80	0.10	46%	31%	Get counter value
0.11	0.00	0.80	0.09	56%	40%	Set counter value
0.91	0.80	1.76	0.10	50%	46%	Tick counter
0.15	0.00	0.96	0.11	46%	34%	Delete counter
0.20	0.16	1.28	0.07	93%	93%	Init flag
0.82	0.64	2.08	0.11	62%	25%	Destroy flag
0.68	0.48	1.44	0.09	68%	9%	Mask bits in flag
0.81	0.64	1.92	0.10	59%	28%	Set bits in flag [no waiters]
1.15	0.96	2.56	0.09	87%	9%	Wait for flag [AND]
1.15	1.12	1.92	0.05	96%	96%	Wait for flag [OR]
1.14	1.12	1.60	0.03	96%	96%	Wait for flag [AND/CLR]
1.14	0.96	1.76	0.06	81%	9%	Wait for flag [OR/CLR]
0.07	0.00	0.16	0.08	53%	53%	Peek on flag
0.50	0.32	1.92	0.11	65%	25%	Create alarm

```

1.60    1.44    4.32    0.21    96%   84% Initialize alarm
0.85    0.64    1.92    0.11    84%    9% Disable alarm
1.47    1.28    3.36    0.17    56%   81% Enable alarm
1.05    0.80    2.56    0.14    87%    6% Delete alarm
0.98    0.80    1.92    0.07    84%    9% Tick counter [1 alarm]
4.74    4.64    5.28    0.09    96%   50% Tick counter [many alarms]
1.74    1.60    3.04    0.11    59%   37% Tick & fire counter [1 alarm]
29.67   29.61   30.09    0.08    71%   71% Tick & fire counters [>1 together]
5.52    5.44    5.92    0.09    96%   59% Tick & fire counters [>1 separately]
5.13    5.12    6.08    0.01    99%   99% Alarm latency [0 threads]
5.64    5.12    6.24    0.36    46%   53% Alarm latency [2 threads]
8.07    6.88    9.76    0.68    43%   37% Alarm latency [many threads]
8.20    8.16   12.49    0.07    98%   98% Alarm -> thread resume latency

1.45    1.28    4.00    0.00                      Clock/interrupt latency

2.13    1.60    5.28    0.00                      Clock DSR latency

6        0      312 (main stack: 1388) Thread stack used (1360 total)
All done, main stack : stack used 1388 size 3920
All done : Interrupt stack used 204 size 4096
All done : Idlethread stack used 796 size 2048

```

Timing complete - 29980 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91SAM9263-EK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The SAM9 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXIX. Atmel AT91RM9200 Processor Support

Overview

Name

Support for the Atmel AT91RM9200 Processor — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Atmel AT91RM9200 processor. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, ARM9 variant HAL and the platform HAL. It provides functionality common to AT91RM9200-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/arm9/at91rm9200` within the eCos source repository.

The AT91RM9200 processor HAL package is loaded automatically when eCos is configured for an AT91RM9200-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Atmel AT91RM9200 processor within this processor HAL package include:

- [AT91RM9200-specific hardware definitions](#)
- [Interrupt controller](#)
- [Timer counters](#)
- [Serial UARTs](#)
- [MultiMedia Card Interface \(MCI\)](#)
- [Two-Wire Interface \(TWI\)](#)
- [Power saving](#)

Support for the on-chip ethernet, interrupt-driven serial, SPI, watchdog and wallclock (RTC) features of the AT91RM9200 are also present and can be found in separate packages, outside of this processor HAL.

The watchdog hardware is also used within this HAL to perform software reset.

Hardware definitions

Name

AT91RM9200 hardware definitions — Details on obtaining hardware definitions for AT91RM9200

Register definitions

The file `<cyg/hal/at91rm9200.h>` can be included from application and eCos package sources to provide definitions related to AT91RM9200 subsystems. These include register definitions for the interrupt controller, power management controller, clock generator, memory controller, external bus interface, GPIO, USART, MCI, TWI (I²C[®]), Ethernet, timer counter, RTC, and SPI subsystems.

Initialization helper macros

The file `<cyg/hal/at91rm9200_init.inc>` contains definitions of helper macros which may be used by AT91RM9200 platform HALs in order to initialise common AT91RM9200 subsystems without excessive duplication between the platform HALs. Typically this file will be included by the `hal_platform_setup.h` header in the platform HAL, in turn included from the architectural HAL file `vectors.S`.

This file is solely intended to be used by platform HALs. At the same time, it is only present to assist initialization, and platform HALs are not obliged to use it if their startup requirements vary.

Interrupt controller

Name

AT91RM9200 interrupt controller — Advanced Interrupt Controller definitions and usage

Interrupt controller definitions

The file <cyg/hal/var_ints.h> (located at hal/arm/arm9/at91rm9200/VERSION/include/var_ints.h in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs:

```
#define CYGNUM_HAL_INTERRUPT_FIQ      0 // Advanced Interrupt Controller (FIQ)
#define CYGNUM_HAL_INTERRUPT_SYSTEM  1 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_INTERRUPT_PIOA     2 // Parallel IO Controller A
#define CYGNUM_HAL_INTERRUPT_PIOB     3 // Parallel IO Controller B
#define CYGNUM_HAL_INTERRUPT_PIOC     4 // Parallel IO Controller C
#define CYGNUM_HAL_INTERRUPT_PIOD     5 // Parallel IO Controller D
#define CYGNUM_HAL_INTERRUPT_US0      6 // USART 0
#define CYGNUM_HAL_INTERRUPT_US1      7 // USART 1
#define CYGNUM_HAL_INTERRUPT_US2      8 // USART 2
#define CYGNUM_HAL_INTERRUPT_US3      9 // USART 3
#define CYGNUM_HAL_INTERRUPT_MCI      10 // Multimedia Card Interface
#define CYGNUM_HAL_INTERRUPT_UDP      11 // USB Device Port
#define CYGNUM_HAL_INTERRUPT_TWI      12 // Two-Wire Interface
#define CYGNUM_HAL_INTERRUPT_SPI      13 // Serial Peripheral Interface
#define CYGNUM_HAL_INTERRUPT_SSC0     14 // Serial Synchronous Controller 0
#define CYGNUM_HAL_INTERRUPT_SSC1     15 // Serial Synchronous Controller 1
#define CYGNUM_HAL_INTERRUPT_SSC2     16 // Serial Synchronous Controller 2
#define CYGNUM_HAL_INTERRUPT_TC0      17 // Timer Counter 0
#define CYGNUM_HAL_INTERRUPT_TC1      18 // Timer Counter 1
#define CYGNUM_HAL_INTERRUPT_TC2      19 // Timer Counter 2
#define CYGNUM_HAL_INTERRUPT_TC3      20 // Timer Counter 3
#define CYGNUM_HAL_INTERRUPT_TC4      21 // Timer Counter 4
#define CYGNUM_HAL_INTERRUPT_TC5      22 // Timer Counter 5
#define CYGNUM_HAL_INTERRUPT_UHP      23 // USB Host port
#define CYGNUM_HAL_INTERRUPT_EMAC     24 // Ethernet MAC
#define CYGNUM_HAL_INTERRUPT_IRQ0     25 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_INTERRUPT_IRQ1     26 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_INTERRUPT_IRQ2     27 // Advanced Interrupt Controller (IRQ2)
#define CYGNUM_HAL_INTERRUPT_IRQ3     28 // Advanced Interrupt Controller (IRQ3)
#define CYGNUM_HAL_INTERRUPT_IRQ4     29 // Advanced Interrupt Controller (IRQ4)
#define CYGNUM_HAL_INTERRUPT_IRQ5     30 // Advanced Interrupt Controller (IRQ5)
#define CYGNUM_HAL_INTERRUPT_IRQ6     31 // Advanced Interrupt Controller (IRQ6)

// The following interrupts are derived from the SYSTEM interrupt
#define CYGNUM_HAL_INTERRUPT_DEBUG    32 // Debug unit
#define CYGNUM_HAL_INTERRUPT_PMC      33 // Power Management Controller
#define CYGNUM_HAL_INTERRUPT_RTCH     34 // Real Time Clock
#define CYGNUM_HAL_INTERRUPT_PIT      35 // System Timer Period Interval Timer
#define CYGNUM_HAL_INTERRUPT_WDOVF    36 // System Timer Watchdog Overflow
#define CYGNUM_HAL_INTERRUPT_RTTINC   37 // System Timer Real-Time Timer Increment
```

```
#define CYGNUM_HAL_INTERRUPT_ALM    38 // System Timer Alarm
```

As indicated above, further decoding is performed on the SYSTEM interrupt to identify the cause more specifically. Note that as a result, placing an interrupt handler on the SYSTEM interrupt will not work as expected. Conversely, masking a decoded derivative of the SYSTEM interrupt will not work as this would mask other SYSTEM interrupts, but masking the SYSTEM interrupt itself will work. On the other hand, unmasking a decoded SYSTEM interrupt *will* unmask the SYSTEM interrupt as a whole, thus unmasking interrupts for the other units on this shared interrupt.

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller functions

The source file `src/at91rm9200_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_irq_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

Interrupts are configured in the `hal_interrupt_configure` function, where the `level` and `up` arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
1	0	Low Level
1	1	High Level

To fit into the eCos interrupt model, interrupts essentially must be acknowledged immediately once decoded, and as a result, the `hal_interrupt_acknowledge` function is empty.

The `hal_interrupt_set_level` is used to set the priority level of the supplied interrupt within the Advanced Interrupt Controller.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Using the Advanced Interrupt Controller for VSRs

The AT91RM9200 HAL has been designed to exploit benefits of the on-chip Advanced Interrupt Controller (AIC) on the AT91RM9200. Support has been included for exploiting its ability to provide hardware vectoring for VSR interrupt handlers.

This support is dependent on definitions that may only be provided by the platform HAL and therefore is only enabled if the platform HAL package implements the `CYGINT_HAL_AT91RM9200_AIC_VSR` CDL interface. The

necessary definitions are available to all platform HALs which use the facilities of the [at91rm9200_init.inc](#) header file.

The interrupt decoding path has been optimised by allowing the AIC to be interrogated for the interrupt handler VSR to use. These vectored interrupts are by default still configured to point to the default ARM architecture HAL IRQ and FIQ VSRs. However applications may set their own VSRs to override this default behaviour to allow optimised interrupt handling.

The VSR vector numbers to use when overriding are defined as follows:

```
#define CYGNUM_HAL_VECTOR_FIQ      7 // FIQ
#define CYGNUM_HAL_VECTOR_SYSTEM  8 // System Peripheral (debug unit, system timer)
#define CYGNUM_HAL_VECTOR_PIOA    9 // Parallel IO Controller A
#define CYGNUM_HAL_VECTOR_PIOB   10 // Parallel IO Controller B
#define CYGNUM_HAL_VECTOR_PIOC   11 // Parallel IO Controller C
#define CYGNUM_HAL_VECTOR_PIOD   12 // Parallel IO Controller D
#define CYGNUM_HAL_VECTOR_US0    13 // USART 0
#define CYGNUM_HAL_VECTOR_US1    14 // USART 1
#define CYGNUM_HAL_VECTOR_US2    15 // USART 2
#define CYGNUM_HAL_VECTOR_US3    16 // USART 3
#define CYGNUM_HAL_VECTOR_MCI    17 // Multimedia Card Interface
#define CYGNUM_HAL_VECTOR_UDP    18 // USB Device Port
#define CYGNUM_HAL_VECTOR_TWI    19 // Two-Wire Interface
#define CYGNUM_HAL_VECTOR_SPI    20 // Serial Peripheral Interface
#define CYGNUM_HAL_VECTOR_SSC0   21 // Serial Synchronous Controller 0
#define CYGNUM_HAL_VECTOR_SSC1   22 // Serial Synchronous Controller 1
#define CYGNUM_HAL_VECTOR_SSC2   23 // Serial Synchronous Controller 2
#define CYGNUM_HAL_VECTOR_TC0    24 // Timer Counter 0
#define CYGNUM_HAL_VECTOR_TC1    25 // Timer Counter 1
#define CYGNUM_HAL_VECTOR_TC2    26 // Timer Counter 2
#define CYGNUM_HAL_VECTOR_TC3    27 // Timer Counter 3
#define CYGNUM_HAL_VECTOR_TC4    28 // Timer Counter 4
#define CYGNUM_HAL_VECTOR_TC5    29 // Timer Counter 5
#define CYGNUM_HAL_VECTOR_UHP    30 // USB Host port
#define CYGNUM_HAL_VECTOR_EMAC   31 // Ethernet MAC
#define CYGNUM_HAL_VECTOR_IRQ0   32 // Advanced Interrupt Controller (IRQ0)
#define CYGNUM_HAL_VECTOR_IRQ1   33 // Advanced Interrupt Controller (IRQ1)
#define CYGNUM_HAL_VECTOR_IRQ2   34 // Advanced Interrupt Controller (IRQ2)
#define CYGNUM_HAL_VECTOR_IRQ3   35 // Advanced Interrupt Controller (IRQ3)
#define CYGNUM_HAL_VECTOR_IRQ4   36 // Advanced Interrupt Controller (IRQ4)
#define CYGNUM_HAL_VECTOR_IRQ5   37 // Advanced Interrupt Controller (IRQ5)
#define CYGNUM_HAL_VECTOR_IRQ6   38 // Advanced Interrupt Controller (IRQ6)
```

Consult the kernel and generic HAL documentation for more information on VSRs and how to set them.

Interrupt handling within standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's

symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers [above](#). Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the [interrupt vector number](#). The `data` argument is extracted from a corresponding table named `hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

Timer counters

Name

Timer counters — Use of on-chip timer counters

Timer counter 0

The eCos kernel system clock is implemented using Timer Counter 0. By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to `CYGNUM_HAL_RTC_DENOMINATOR`, then the configuration option `CYGNUM_HAL_RTC_NUMERATOR` may also be adjusted, and again clock-related settings will automatically be recalculated.

Timer Counter 0 is also used to implement the HAL microsecond delay function, `HAL_DELAY_US`. This is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Timer-based profiling support

Timer-based profiling support is implemented using timer counter 1 (TC1). If the `gprof` package, `CYGPKG_PROFILE_GPROF`, is included in the configuration, then TC1 is reserved for use by the profiler.

Serial UARTs

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

Support is included in this processor HAL package for the AT91RM9200's on-chip debug unit UART and four serial USART serial devices.

There are two forms of support: HAL diagnostic I/O; and a fully interrupt-driven serial driver. Unless otherwise specified in the platform HAL documentation, for all serial ports the default settings are 115200,8,N,1 with no flow control.

HAL diagnostic I/O

This first form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus this mode is not usually suitable for deployed systems. This can operate on any port, according to the configuration settings.

There are several configuration options usually found within a platform HAL which affect the use of this support in the AT91RM9200 processor HAL. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on any port.

Note that it is not recommended to share this driver when using the HAL diagnostic I/O on the same port. If the driver is shared with the GDB debugging port, it will prevent ctrl-c operation when debugging.

This driver is contained in the `CYGPKG_IO_SERIAL_ARM_AT91` package. That driver package should also be consulted for documentation and configuration options. The driver is not active until the `CYGPKG_IO_SERIAL_DEVICES` configuration option within the generic serial driver support package `CYGPKG_IO_SERIAL` is enabled in the configuration.

Note that unlike the USART devices, the serial debug port does not support modem control signals such as those used for hardware signals. In addition, USART devices for a particular platform may also not have these control signals brought out to the physical serial port.

Multimedia Card Interface (MCI) driver

Name

Multimedia Card Interface (MCI) driver — Using MMC/SD cards with block drivers and filesystems

Overview

The MultiMedia Card Interface (MCI) driver in the AT91RM9200 processor HAL allows use of MultiMedia Cards (MMC cards) and Secure Digital (SD) flash storage cards within eCos, exported as block devices. This makes them suitable for use as the underlying devices for filesystems such as FAT.

Configuration

This driver provides the necessary support for the generic MMC/SD bus layer within the `CYGPKG_DEVS_DISK_MMC` package to export a disk block device. The disk block device is only available if the generic disk I/O layer found in the package `CYGPKG_IO_DISK` is included in the configuration.

The block device may then be used as the device layer for a filesystem such as FAT. Example devices are `"/dev/mmc0/1"` to refer to the first partition on the card, or `"/dev/mmc0/0"` to address the whole device including potentially the partition table at the start.

The driver may be forcibly disabled within this processor HAL package with the configuration option `CYGPKG_HAL_ARM_ARM9_AT91RM9200_MCI`.

If the driver is enabled, there are only two AT91RM9200 specific options:

`CYGIMP_HAL_ARM_ARM9_AT91RM9200_MCI_INTMODE`

This indicates that the driver should operate in interrupt-driven mode if possible. This is enabled by default if the eCos kernel is enabled. Note though that if the driver finds that global interrupts are off when running, then it will fall back to polled mode even if this option is enabled. This allows for use of the MCI driver in an initialisation context.

`CYGNUM_HAL_ARM_ARM9_AT91RM9200_MCI_POWERSAVE_DIVIDER`

The AT91RM9200 MCI peripheral allows the MCI clock to be divided down if told to enter power saving mode. This option specifies the divider to use. The driver itself does not implement any power saving - it is up to the application to enable power saving in the MCI control register if it is required.

Usage notes

MMC/SD cards may only be used in a MMC/SD card slot, and not a dataflash slot. The driver will detect the appropriate card sizes. Hotswapping of cards is supported by the driver, and in the case of eCosPro, the FAT

Multimedia Card Interface (MCI) driver

filesystem. Although any cards removed before explicit unmounting or a `sync()` call to flush filesystem buffers will likely result in a corrupted filesystem on the removed card.

The MMC/SD bus layer will parse partition tables, although it can be configured to allow handling of cards with no partition table.

Two-Wire Interface (TWI) driver

Name

Two-Wire Interface (TWI) driver — Configuration and implementation details of TWI (I²C®) driver

Overview

The AT91RM9200 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91RM9200. This type of bus is also known as I²C®. The API for this may be found within the `CYGPKG_IO_I2C` package.

I²C®/TWI driver configuration

The I²C® driver uses the AT91RM9200's internal Two-Wire Interface (TWI) support. This is controlled within the AT91RM9200 processor HAL (`CYGPKG_HAL_AT91RM9200`). The `CYGPKG_HAL_AT91RM9200_TWI` CDL component controls whether the TWI driver is enabled. Within that component, there are two sub-options:

- `CYGNUM_HAL_AT91RM9200_TWI_CLOCK` sets the speed of the TWI bus clock in Hz. This is usually 100kHz, but can be set up to 400kHz if the devices on the bus support this speed, also known as fast mode. However other values below 400kHz can also be chosen, subject to the accuracy of the clock waveform generation parameters.
- The second option within the `CYGPKG_HAL_AT91RM9200_TWI` component is `CYGNUM_HAL_AT91RM9200_TWI_CKDIV`. This is the clock divider used when configuring the `TWI_CWGR` register. Consult the AT91RM9200 datasheet description of the `TWI_CWGR` register for the formula used to determine the clock frequency. Increasing the divider will decrease the accuracy in practice of the generated I²C bus clock compared to `CYGNUM_HAL_AT91RM9200_TWI_CLOCK`. But the divider must also be sufficiently low that the relevant factors do not overflow valid values for `CHDIV/CLDIV` in `TWI_CWGR`. Note that when the AT91RM9200 is using a 60MHz MCK, then for 100kHz operation, a value for this option of 1 is most appropriate. For 400kHz, a value for this option of 0 is most appropriate. The default value of this CDL is an appropriate value for `CKDIV` assuming a 60MHz MCK and a TWI clock between 29kHz and 400kHz.

To be specific, the `CLDIV/CHDIV` fields of the `TWI_CWGR` are considered equal. The value of, for example, `CLDIV`, can be expressed as:

$$CLDIV = \frac{f_{MCK} - 6f_{TWI}}{2^{CKDIV+1} \cdot f_{TWI}}$$

To use the I²C/TWI driver, the generic I²C driver package `CYGPKG_IO_I2C` must be used. Documentation for its API may be found elsewhere.

Usage notes

This driver only operates in interrupt mode. It does not operate in polled mode, and thus does not operate when interrupts are disabled. It cannot therefore be used in an initialization context, before the eCos kernel thread scheduler starts. And it cannot be used with RedBoot.

Due to the characteristics of the AT91RM9200's operation, it is not possible to provide support for repeated starts with the I²C package API. Similarly indicating a NACK when performing a receive is equivalent to also sending a STOP.

A test application for use with the Aardvark I²C/SPI Activity Board (<http://www.totalphase.com/products/accessories/activity-board/>) is provided within the `tests` subdirectory of the `CYGPKG_HAL_AT91RM9200` package. This test communicates with the I²C EEPROM on the board to perform read and write operations using I²C. This test is not built by default. It may be built by enabling the configuration option `CYGBLD_HAL_ARM_ARM9_AT91RM9200_TEST_TWI_AT24C02A` within the AT91RM9200 processor HAL.

Power saving support

Name

Power saving support — Extensions for saving power

Overview

There is support in the AT91RM9200 processor HAL for a simple power saving mechanism. This is provided by two functions:

```
#include <cyg/hal/hal_intr.h>

__externC void cyg_hal_at91rm9200_powersave_init( cyg_uint32 ip_addr );

__externC void cyg_hal_at91rm9200_powerdown( void );
```

The powersaving system is initialized by calling `cyg_hal_at91rm9200_powersave_init()`. The argument should be the IP address of this machine in network order. This can usually be fetched from the bootp data for an interface after completion of the call to `init_all_network_interfaces()`. e.g. `eth0_bootp_data.bp_ciaddr.s_addr`.

A call to `cyg_hal_at91rm9200_powerdown()` will put the machine into a low power mode. This will involve switching to a slower system clock speed, disabling all peripherals except those that are defined to cause the system to wake up and return from this function.

Configuration

The exact behaviour of the power saving system is controlled by the following configuration options:

CYGPKG_HAL_ARM_ARM9_AT91RM9200_POWERSAVE

This option controls the overall inclusion of the power saving system.

Default value: on

CYGSEM_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_POLL_ETHERNET

This option enables polling of the ethernet interface for relevant ARP packets and unicast IP packets. It is necessary for the CPU to run at a higher CPU speed for this option to work.

Default value: off

CYGSEM_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_IDLE

If this option is set, the CPU will go into idle mode, which will cause it to halt until an interrupt is delivered.

Default value: off

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_ACTIVE_DEVICES

This option defines the devices that are to be kept running during power down mode. An interrupt from one of these devices is usually the only way of bringing the system out of idle mode. The value of this option is a bit mask with bits set for each device that is to be kept active. The bits correspond to the peripheral identifiers described in the AT91RM9200 documentation.

Default value: 0x00000000

CYGSEM_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_POLL_GPIO

This option control whether the power saving system will poll GPIO pins during power saving. For this to work the CPU cannot be put into idle mode.

Default value: on

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_PIO_HI

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 1, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_PIO_LO

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to be 0, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGVAR_HAL_ARM_ARM9_AT91RM9200_POWERSAVE_PIO_CHANGE

This is an array of bitmasks of the bits in the PIO PDSR registers. Within the array, index 0 corresponds to PIOA, index 1 to PIOB and so on. For each set bit in these masks, if the value is seen to change between successive polls, then the low power mode will be terminated.

Default value: 0, 0, 0, 0

CYGBLD_HAL_ARM_ARM9_AT91RM9200_TEST_POWERSAVE

This option controls whether a simple test is built to exercise power saving support. The test is not built by default as an external means is required to wake the processor up by one of the above configured mechanisms.

Default value: 0

XXX. Atmel AT91RM9200 Development Kit/Evaluation Kit Board Support

Overview

Name

eCos Support for the Atmel AT91RM9200 Development Kit/Evaluation Kit —
Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the Atmel AT91RM9200 Development Kit and Atmel AT91RM9200 Evaluation Kit. The AT91RM9200 Evaluation Kit (EK board) contains the AT91RM9200 processor, 8Mbytes of SDRAM, 8Mbytes of parallel NOR flash memory, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for two serial channels (one debug, one full), ethernet, USB host/device, graphics, and the various other peripherals supported by the AT91RM9200. The AT91RM9200 Development Kit (DK board) is similar but also comes with a 128Kbytes TWI (I2C) EEPROM, IrDA port, and 8Mbytes of SPI DataFlash, although only 2Mbytes of parallel NOR flash memory. eCos support for the many devices and peripherals on the boards and the AT91RM9200 is described below.

In this document, an EK board will be assumed for the purposes of examples and output.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the AT91RM9200 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

On the EK board, the parallel NOR flash memory consists of 8 blocks of 8Kbytes each, followed by 127 blocks of 64Kbytes each. In a typical setup, the first 192 Kbytes are reserved for the use of the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 125 blocks available between 0x60030000 and 0x607EFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The debug serial port at J10 and DTE port at J14 (connected to USART channel 1) can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the AT91RM9200-EK or AT91RM9200-DK targets.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the EK/DK hardware. This driver is also loaded automatically when configuring for the EK or DK targets.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200`. This driver is also loaded automatically when configuring for the EK or DK targets.

There is a driver for the on-chip real-time clock (RTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91`. This driver is also loaded automatically when configuring for the EK or DK targets.

The AT91RM9200 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91RM9200. This type of bus is also known as I²C®. Further documentation may be found in the AT91RM9200 processor HAL documentation.

The AT91RM9200 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the EK or DK targets and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the AT91RM9200 processor HAL documentation.

There is a driver `CYGPKG_DEVS_SPI_ARM_ATMEL_AT91RM9200_KITS` to allow access to devices on the SPI bus. This driver provides information to the more general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which in turn provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the EK or DK targets.

Furthermore, the platform HAL package contains support for SPI dataflash cards. The HAL support integrates with the `CYGPKG_DEVS_FLASH_ATMEL_DATAFLASH` package as well as the above SPI packages. That package is automatically loaded when configuring for the EK or DK targets. Dataflash media is then accessed as a Flash device, using the Flash I/O API within the `CYGPKG_IO_FLASH` package, if that package is loaded in the configuration.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The AT91RM9200-EK and AT91RM9200-DK support is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the AT91RM9200-EK and AT91RM9200-DK boards for eCos Development

Overview

In a typical development environment, the AT91RM9200-EK/DK boards boot from the parallel NOR Flash and run the RedBoot ROM monitor directly. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.bin
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.bin
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	redboot_ROMRAM.ecm	redboot_ROMRAM.bin
JTAG	RedBoot running from RAM, loaded via JTAG	redboot_JTAG.ecm	redboot_JTAG.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

The ample provision of RAM memory on the board allows the ROMRAM version of RedBoot to be preferential to the standard ROM version which executes directly from Flash. Alternatively, if the ROM version is to be chosen, then the RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to in turn load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged.

Initial Installation

The on-chip boot program on the AT91RM9200 is only capable of loading programs into 12Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently two mechanisms are described below to program RedBoot into Flash. Both of them require a JTAG device. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.at91rm9200ek.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/arm9/atmel-at91rm9200-kits/VERSION/misc` relative to the root of your eCos installation.
5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the `bdi2000.at91rm9200ek.cfg` in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.
8. Open `bdi2000.at91rm9200ek.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.at91rm9200ek.cfg` configuration file at the appropriate point of this process.

Preparing the AT91RM9200-EK/DK board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between the Serial Debug Port on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to your host PC's LAN with an Ethernet cable.
4. You should designate the board with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.
5. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the ICE interface connector to the Target A port on the BDI2000.
6. Locate jumper J15 on the board, which is by default set to `INT`. It should be reset to `EXT`. In due course this will ensure that the board boots RedBoot from the external parallel Flash device.
7. Power up the AT91RM9200-EK board. You should see the three ethernet LEDs illuminate.
8. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

Core#0>

9. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
Core#0> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x05B0203F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

10. Locate the redboot_ROMRAM.bin image within the loaders subdirectory of the base of the eCos installation.
11. Copy the redboot_ROMRAM.bin file into a location on the host computer accessible to its TFTP server.

Method 1 - Using the BDI2000 to directly program RedBoot into Flash

As previously mentioned, there are two methods of programming a RedBoot image into the parallel NOR Flash. This method uses the built-in capabilities of the BDI2000.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.
2. Cut and paste the following commands into the BDI2000 telnet session. They are used to unlock the relevant Flash blocks that will contain RedBoot. The BDI2000 does have an **unlock** command, however this only works with Intel StrataFLASH and is therefore not suitable.

```
mmh 0x1000aaaa 0x00aa
mmh 0x10000000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10002000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10004000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10006000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10008000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x1000a000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x1000c000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x1000e000 0x0070
mmh 0x1000aaaa 0x00aa
mmh 0x10010000 0x0070
```

```
mmh 0x1000aaaa 0x00aa
mmh 0x10020000 0x0070
```

3. Erase the 8 initial 8Kbyte sized Flash blocks, and the following 2 64Kbyte Flash blocks with the following commands:

```
Core#0>erase 0x10000000 0x2000 8
Erasing flash at 0x10000000
Erasing flash at 0x10002000
Erasing flash at 0x10004000
Erasing flash at 0x10006000
Erasing flash at 0x10008000
Erasing flash at 0x1000a000
Erasing flash at 0x1000c000
Erasing flash at 0x1000e000
Erasing flash passed
Core#0>erase 0x10010000 0x10000 2
Erasing flash at 0x10010000
Erasing flash at 0x10020000
Erasing flash passed
Core#0>
```

4. Program the RedBoot image into Flash with the following command, replacing */RBPATH* with the location of the *redboot_ROMRAM.bin* file relative to the TFTP server root directory:

```
Core#0>prog 0x10000000 /RBPATH/redboot_ROMRAM.bin bin
Programming /RBPATH/redboot_ROMRAM.bin , please wait ...
Programming flash passed
Core#0>
```

This operation can take some time.

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. The RedBoot banner should be visible on the serial port. RedBoot's Flash configuration can be initialized using the [same procedure as required in Method 2 below](#).

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into Flash with RAM RedBoot

With this approach, the BDI2000 is used to load a RAM RedBoot image, which can then in turn be used to load and program a ROMRAM RedBoot image into Flash.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot.

Loading a RAM RedBoot

1. Locate the *redboot_JTAG.bin* image within the *loaders* subdirectory of the base of the eCos installation.
2. Copy the *redboot_JTAG.bin* file into a location on the host computer accessible to its TFTP server.

3. With the BDI2000 telnet interface, execute the following command, replacing */RBPATH* with the location of the redboot_JTAG.bin file relative to the TFTP server root directory:

```
Core#0>load 0x20008000 /RBPATH/redboot_JTAG.bin bin
Loading /RBPATH/redboot_JTAG.bin , please wait ....
Loading program file passed
Core#0>
```

4. Run the loaded RAM RedBoot:

```
Core#0>go 0x20008000
Core#0>
```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
***Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
PHY: Davicom DM9161A
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb/Full Duplex
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 192.168.7.190/255.255.255.0, Gateway: 192.168.7.1
Default server: 192.168.7.11, DNS server IP: 192.168.7.11

RedBoot(tm) bootstrap and debug environment [RAM]
eCosCentric certified release, version v2_XX - built 18:51:18, Aug 25 2005

Platform: Atmel AT91RM9200-EK (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

RAM: 0x20000000-0x20800000, [0x2002f4e0-0x207ed000] available
FLASH: 0x60000000 - 0x607ffffff 8 x 0x2000 blocks 127 x 0x10000 blocks
RedBoot>
```

In the above output, a local BOOTP/DHCP server was able to serve an address to the device.

Note: It is also possible to use the RAM startup version of RedBoot and the redboot_RAM.bin file instead of redboot_JTAG.bin above. If so, then the address to the **load** command must be 0x20040000, as must be the address to the **go** command.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash, but is also applicable to initial configuration of a ROMRAM RedBoot loaded using [Method 1](#).

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```

RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x60030000-0x607effff: .....
... Unlocking from 0x607f0000-0x607fffff: .
... Erase from 0x607f0000-0x607fffff: .
... Program from 0x207f0000-0x20800000 to 0x607f0000: .
... Locking from 0x607f0000-0x607fffff: .
RedBoot>

```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```

Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```

RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91rm9200_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x607f0000-0x607fffff: .
... Erase from 0x607f0000-0x607fffff: .
... Program from 0x207f0000-0x20800000 to 0x607f0000: .
... Locking from 0x607f0000-0x607fffff: .
RedBoot>

```

Loading and programming the ROMRAM RedBoot

This section describes the steps required to load the ROMRAM RedBoot from the TFTP server and program it into Flash.

1. Load the RedBoot ROMRAM binary image from the TFTP server. Use the following command, replacing *111.222.333.444* with the TFTP server IP address (or domain name if a DNS server has been configured), and */RBPATH* with the location of the `redboot_ROMRAM.bin` file relative to the TFTP server root directory:

```

RedBoot> load -r -b {%freememlo} -h 111.222.333.444 /RBPATH/redboot_ROMRAM.bin
Using default protocol (TFTP)
Raw file loaded 0x2002f800-0x2004e367, assumed entry at 0x2002f800
RedBoot>

```

2. Finally install the loaded image into Flash:

```

RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Unlocking from 0x60000000-0x6002ffff: .....
... Erase from 0x60000000-0x6002ffff: .....
... Program from 0x0002f800-0x0004e368 to 0x60000000: .....
... Locking from 0x60000000-0x6002ffff: .....
... Unlocking from 0x607f0000-0x607fffff: .
... Erase from 0x607f0000-0x607fffff: .
... Program from 0x007f0000-0x00800000 to 0x607f0000: .
... Locking from 0x607f0000-0x607fffff: .
RedBoot>

```

It is also possible to use the **fis write** command to write the image into Flash, but if so, the relevant Flash blocks must also be explicitly unlocked with the command:

```

RedBoot> fis unlock -f 0x60000000 -l 0x30000

```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port. Verify the IP settings are as expected.

```

+PHY: Davicom DM9161A
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb/Full Duplex
Ethernet eth0: MAC address 00:23:31:37:00:3d
IP: 192.168.7.222/255.255.255.0, Gateway: 192.168.7.1
Default server: 192.168.7.9, DNS server IP: 192.168.7.11

```

```

RedBoot(tm) bootstrap and debug environment [ROMRAM]
eCosCentric certified release, version v2_XX - built 18:55:20, Aug 25 2005

```

```

Platform: Atmel AT91RM9200-EK (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005 eCosCentric Limited

```

```

RAM: 0x20000000-0x20800000, [0x20030470-0x207ed000] available
FLASH: 0x60000000 - 0x607fffff 8 x 0x2000 blocks 127 x 0x10000 blocks
RedBoot>

```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the ROMRAM version of RedBoot for the AT91RM9200-EK are:

```
$ mkdir redboot_at91rm9200ek_romram
$ cd redboot_at91rm9200ek_romram
$ ecosconfig new at91rm9200ek redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/atmel-at91rm9200-kits/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - ROM, RAM or JTAG - may be similarly built by choosing the appropriate alternative `.ecm` file.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The AT91RM9200-EK/DK platform HAL package is loaded automatically when eCos is configured for `at91rm9200ek` or `at91rm9200dk` targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x10000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location `0x10000000`. However, when it starts up, the application will first copy itself to RAM at virtual address `0x00000000` and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The AT91RM9200-EK board contains an 8Mbyte Atmel AT49BV6416 parallel Flash device. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the AT91RM9200-EK board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal EMAC ethernet device attached to an external Davicom DM9161A PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the AT91RM9200-EK/DK boards. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91RM9200_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

USART Serial Driver

The AT91RM9200-EK/DK boards use the AT91RM9200's internal USART serial support as described in the AT91RM9200 processor HAL documentation. Two serial ports are available: the serial debug port which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/dbg"` in the interrupt-driven driver; and USART 1 which is mapped to virtual vector channel 1 and `"/dev/ser1"`. Only USART 1 supports modem control signals such as those used for hardware flow control.

MCI Driver

As the AT91RM9200 MCI driver is part of the AT91RM9200 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

Various options can be used to control specific of the AT91RM9200 MCI driver. Consult the AT91RM9200 HAL documentation for information on its configuration.

On this target, the MMC/SD socket allows detection of when cards are inserted and removed. This may be used with the removeable media support and disk insertion/removal event notification system in the disk I/O package so that the application or other eCos subsystems are informed when cards are inserted and removed. This in turn allows use of the automounter contained within the File I/O package (CYGPKG_FILEIO) to mount and unmount cards automatically.

Caution

Remember that the ability to unmount cards after removal does not prevent those cards containing corrupt filesystems - instead cards should be preferably unmounted before removal, or at least have the filesystem's in-memory buffers flushed to the media using the `sync()` function).

The MMC/SD socket also allows detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will be detected and mounted read-only, and attempts to write to them will fail.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM920T core of the AT91RM9200 only supports two such hardware breakpoints, and so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.at91rm9200ek.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `bdi2000.at91rm9200ek.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the `bdiGDB` interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.at91rm9200ek.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROMRAM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
Core#0>load 0x20008000 /test.bin bin
Loading /test.bin , please wait ....
```

```
Loading program file passed  
Core#0>go 0x20008000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the SDRAM, such as by using the `bdi2000.at91rm9200ek.cfg` configuration on the BDI2000, RAM applications can be loaded directly into SDRAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. This will also cause the `CYGBLD_HAL_ARM9_ATMEL_AT91RM9200_KITS_LOAD_LOW_RAM` configuration option to be enabled allowing the application to be built with a set of memory layout files that will configure the linker script to set the program load address to be within the physical SDRAM space. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be disabled in order to prevent HAL diagnostic output being encoded into GDB (\$) packets.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

Warning

If resetting the board using the JTAG device, such as by using the BDI2000 **reset** command, the Ethernet PHY fails to interface correctly with the AT91RM9200, and consequently all subsequent ethernet operations are impossible. Only a reset by pressing the reset button or due to a watchdog timeout will cause the PHY to reset correctly.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the AT91RM9200-EK/DK hardware, and should be read in conjunction with that specification. The AT91RM9200-EK/DK platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the AT91RM9200 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. The HAL uses the MMU to locate it at virtual address 0x60000000 after initialization. It remains accessible at address 0x10000000 but accesses to this address range are uncached.

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

SPI dataflash

SPI Dataflash media can only be accessed with the Flash API. For the purposes of this API a placeholder address range has been allocated as if the Flash is present at this address. The base of this address range is 0x30000000 and the extent will clearly depend on the Dataflash capacity. This reserved range is not real memory and any attempt to access it directly by the processor other than via the Flash API will result in a memory address exception.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, parallel NOR flash, ethernet PHY, SPI dataflash and MCI facilities on the AT91RM9200-EK/DK boards. eCos does not currently make any use of any other off-chip peripherals present on these boards.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91RM9200, using the facilities of the AT91RM9200 processor HAL. Consult the documentation in that package for details.

SPI Dataflash

eCos supports SPI access to Dataflash on the AT91RM9200. Two physical slots are provided on the board, but only the upper one may be used for SPI dataflash, not the one on the underside. This is due to an AT91RM9200 errata affecting SPI chip selects.

Accesses to Dataflash are performed via the Flash API, using 0x30000000 as the nominal address of the device, although it does not truly exist in the processor address space. On driver initialisation, eCos and RedBoot can

detect the presence of a card in the socket. In particular, on reset RedBoot will indicate the presence of Flash at the 0x30000000 address range in its startup banner if it has been successfully detected. Hot swapping is not possible.

Since Dataflash is not directly addressable, access from RedBoot is only possible using **fis** command operations. Flash partitions within the FIS can be created, although users should be aware that the FIS partition data is stored in the NOR flash, and not on a per-Dataflash card basis. Therefore if a second Dataflash card is inserted it will appear to have the same FIS partitions residing on the card. Care must be taken if swapping between cards with differing partition layouts.

The MCI driver cannot be enabled simultaneously with the SPI driver, as the drivers need differing pin configurations for the same pins on this board due to the shared socket.

Real-time characterization

The `tm_basic` kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    412 size  3920
Startup : Interrupt stack used    524 size  4096
Startup : Idlethread stack used    80 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead

... this value will be factored out of all other measurements

Clock interrupt took 13.02 microseconds (6 raw clock ticks)

Testing parameters:

```
Clock samples:      32
Threads:            64
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence			
Ave	Min	Max	Var	Ave	Min	Function	
=====	=====	=====	=====	=====	=====	=====	
14.09	8.55	19.23	2.46	45%	25%	Create thread	
2.10	0.00	4.27	0.13	95%	3%	Yield thread [all suspended]	
2.74	2.14	6.41	0.88	73%	73%	Suspend [suspended] thread	
2.54	2.14	4.27	0.65	81%	81%	Resume thread	
3.30	2.14	6.41	1.09	51%	46%	Set priority	
0.47	0.00	2.14	0.73	78%	78%	Get priority	
7.24	6.41	14.96	1.09	65%	65%	Kill [suspended] thread	
2.10	0.00	4.27	0.13	95%	3%	Yield [no other] thread	

4.21	2.14	6.41	0.19	93%	4%	Resume [suspended low prio] thread
2.40	2.14	4.27	0.47	87%	87%	Resume [runnable low prio] thread
2.94	2.14	4.27	1.00	62%	62%	Suspend [runnable] thread
2.14	0.00	4.27	0.07	96%	1%	Yield [only low prio] thread
2.17	2.14	4.27	0.06	98%	98%	Suspend [runnable->not runnable]
7.08	6.41	12.82	0.96	71%	71%	Kill [runnable] thread
5.41	4.27	8.55	1.10	50%	48%	Destroy [dead] thread
10.32	8.55	14.96	0.72	78%	20%	Destroy [runnable] thread
12.69	10.68	19.23	0.81	70%	20%	Resume [high priority] thread
5.56	4.27	8.55	1.04	58%	40%	Thread switch
0.52	0.00	2.14	0.78	75%	75%	Scheduler lock
1.77	0.00	2.14	0.61	82%	17%	Scheduler unlock [0 threads]
1.77	0.00	2.14	0.61	82%	17%	Scheduler unlock [1 suspended]
1.77	0.00	2.14	0.61	82%	17%	Scheduler unlock [many suspended]
1.78	0.00	4.27	0.61	82%	17%	Scheduler unlock [many low prio]
0.87	0.00	2.14	1.03	59%	59%	Init mutex
2.74	2.14	6.41	0.90	75%	75%	Lock [unlocked] mutex
3.07	2.14	4.27	1.05	56%	56%	Unlock [locked] mutex
2.34	2.14	4.27	0.36	90%	90%	Trylock [unlocked] mutex
2.14	2.14	2.14	0.00	100%	100%	Trylock [locked] mutex
0.53	0.00	2.14	0.80	75%	75%	Destroy mutex
12.89	12.82	14.96	0.13	96%	96%	Unlock/Lock mutex
0.80	0.00	4.27	1.05	65%	65%	Create mbox
0.80	0.00	2.14	1.00	62%	62%	Peek [empty] mbox
2.60	2.14	4.27	0.73	78%	78%	Put [first] mbox
0.33	0.00	2.14	0.56	84%	84%	Peek [1 msg] mbox
2.67	2.14	4.27	0.80	75%	75%	Put [second] mbox
0.53	0.00	2.14	0.80	75%	75%	Peek [2 msgs] mbox
2.60	2.14	4.27	0.73	78%	78%	Get [first] mbox
2.47	2.14	4.27	0.56	84%	84%	Get [second] mbox
2.47	2.14	4.27	0.56	84%	84%	Tryput [first] mbox
2.20	2.14	4.27	0.13	96%	96%	Peek item [non-empty] mbox
2.54	2.14	4.27	0.65	81%	81%	Tryget [non-empty] mbox
2.07	0.00	2.14	0.13	96%	3%	Peek item [empty] mbox
2.20	2.14	4.27	0.13	96%	96%	Tryget [empty] mbox
0.47	0.00	2.14	0.73	78%	78%	Waiting to get mbox
0.47	0.00	2.14	0.73	78%	78%	Waiting to put mbox
2.67	2.14	6.41	0.83	78%	78%	Delete mbox
6.61	6.41	12.82	0.38	96%	96%	Put/Get mbox
0.73	0.00	2.14	0.96	65%	65%	Init semaphore
2.20	0.00	4.27	0.26	90%	3%	Post [0] semaphore
2.20	2.14	4.27	0.13	96%	96%	Wait [1] semaphore
2.00	0.00	2.14	0.25	93%	6%	Trywait [0] semaphore
2.07	0.00	2.14	0.13	96%	3%	Trywait [1] semaphore
0.73	0.00	2.14	0.96	65%	65%	Peek semaphore
0.33	0.00	2.14	0.56	84%	84%	Destroy semaphore
7.28	6.41	10.68	1.08	62%	62%	Post/Wait semaphore
1.00	0.00	2.14	1.06	53%	53%	Create counter
0.93	0.00	2.14	1.05	56%	56%	Get counter value
0.60	0.00	2.14	0.86	71%	71%	Set counter value

2.74	2.14	4.27	0.86	71%	71%	Tick counter			
0.53	0.00	2.14	0.80	75%	75%	Delete counter			
0.73	0.00	2.14	0.96	65%	65%	Init flag			
2.54	2.14	4.27	0.65	81%	81%	Destroy flag			
2.00	0.00	4.27	0.38	87%	9%	Mask bits in flag			
2.40	2.14	4.27	0.47	87%	87%	Set bits in flag [no waiters]			
3.40	2.14	6.41	1.11	53%	43%	Wait for flag [AND]			
3.27	2.14	4.27	1.06	53%	46%	Wait for flag [OR]			
3.27	2.14	4.27	1.06	53%	46%	Wait for flag [AND/CLR]			
3.34	2.14	4.27	1.05	56%	43%	Wait for flag [OR/CLR]			
0.40	0.00	2.14	0.65	81%	81%	Peek on flag			
1.53	0.00	2.14	0.86	71%	28%	Create alarm			
4.41	2.14	8.55	0.51	84%	6%	Initialize alarm			
2.00	0.00	4.27	0.38	87%	9%	Disable alarm			
4.34	4.27	6.41	0.13	96%	96%	Enable alarm			
2.40	2.14	4.27	0.47	87%	87%	Delete alarm			
3.14	2.14	4.27	1.06	53%	53%	Tick counter [1 alarm]			
14.69	12.82	14.96	0.47	87%	12%	Tick counter [many alarms]			
5.01	4.27	6.41	0.96	65%	65%	Tick & fire counter [1 alarm]			
86.34	85.47	87.61	1.03	59%	59%	Tick & fire counters [>1 together]			
16.89	14.96	17.09	0.36	90%	9%	Tick & fire counters [>1 separately]			
10.75	10.68	19.23	0.13	99%	99%	Alarm latency [0 threads]			
13.59	10.68	21.37	1.64	76%	22%	Alarm latency [2 threads]			
13.89	10.68	25.64	1.30	92%	4%	Alarm latency [many threads]			
19.63	19.23	61.97	0.77	96%	96%	Alarm -> thread resume latency			
2.15	2.14	6.41	0.00			Clock/interrupt latency			
5.05	2.14	8.55	0.00			Clock DSR latency			
48	0	296	(main stack: 1408)	Thread stack used (1360 total)					
	All done, main stack : stack used 1408 size 3920								
	All done : Interrupt stack used 208 size 4096								
	All done : Idlethread stack used 732 size 2048								

Timing complete - 29590 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The AT91RM9200-EK/DK platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91RM9200 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXI. Cogent CSB337 Board Support

Overview

Name

eCos Support for the CSB337 Board — Overview

Description

This document covers the Cogent CSB337 single board computer based on the Atmel AT91RM9200. The CSB337 contains the AT91RM9200 processor, 32Mb of SDRAM, 8MB of flash memory, an Intel LXT971 PHY and external connections for two serial channels, ethernet and the various other peripherals supported by the AT91RM9200. The CSB337 is usually plugged into a breakout board such as a Cogent CSB300 or CSB300CF.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 64 blocks of 128k bytes each. In a typical setup, the first flash block is used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 60 blocks between 0x60020000 and 0x607DFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the CSB337 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_CSB337` is responsible for configuring this generic driver to the CSB337 hardware. These drivers are also loaded automatically when configuring for the CSB337 target.

The on-chip TWI device is not supported. Instead there is a bit-banged I2C bus using GPIO pins PA25 and PA26, with one attached device: a DS1307 battery-backed wallclock. The bus is supported by the `CYGPKG_IO_I2C` package and some platform-specific support. The platform HAL provides a `cyg_i2c_bus` structure `hal_csb337_i2c_bus`, and one `cyg_i2c_device` structure `cyg_i2c_wallclock_ds1307`. The wallclock is used mainly by the DS1307 device driver, but it also provides 56 bytes of non-volatile storage which can be used by the application. Any unused I2C functionality will be eliminated at link-time.

`CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1307` provides support for the DS1307 clock. This will be inactive unless the generic wallclock support `CYGPKG_IO_WALLCLOCK` is loaded. Some templates load this automatically, otherwise it must be loaded explicitly. The wallclock is not normally accessed directly. Instead it provides support for the standard C library time-related routines such as `time` and `asctime`, and can be updated by an eCos-specific function `cyg_libc_time_settime`.

eCos manages the on-chip interrupt controller. Timer counter 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (SPI, MCI etc.) are not touched.

Tools

The CSB337 port is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.2.1, arm-elf-gdb version 5.3, and binutils version 2.13.1.

Setup

Name

Setup — Preparing the CSB337 board for eCos Development

Overview

In a typical development environment, the CSB337 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.bin
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector.	redboot_ROMRAM.ecm	redboot_ROMRAM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

The CSB337 boards are shipped from Cogent with a version of Micromonitor installed.

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Micromonitor ROM image. This is a two stage process, you must first download a RAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory.

Connect a serial cable between the CSB337 board serial port 0 and a host computer and start a terminal emulator such as HyperTerminal. Experiments indicate that the version of the Xmodem protocol used by Micromonitor is incompatible with that used by the Linux minicom program. It does work with HyperTerminal, so at present RedBoot must be installed from a Windows host.

When Micromonitor starts up you will see something similar to this:

```
TFS Scanning //FLASH/...
EMAC: Auto-Negotiate Complete, Link = 100MBIT, Full Duplex.
MICRO MONITOR
CPU: AT91RM9200 ARM920T
Platform: Cogent CSB337 - AT91RM9200 SBC
Built: Jan_29,2004 @ 11:42:57
Monitor RAM: 0x20000000-0x2001a044
```

```
Application RAM Base: 0x20100000
MAC: 00:23:31:37:00:01
IP: 192.168.254.210
uMON>
```

Start the download by giving the following command to Micromonitor:

```
uMON>xmodem -d -a 0x20040000
```

You may get a sequence of binary characters, which indicate that Micromonitor is waiting for the download to start. Use HyperTerminal's X-Modem file transfer option to send the file `redboot_RAM.bin`.

When the transfer is finished you will see something like:

```
Rcvd 868 pkts (111104 bytes)
EMAC: Auto-Negotiate Complete, Link = 100MBIT, Full Dupex.
uMON>
```

Start RedBoot with the **call** command, which should result in RedBoot starting up.

```
uMON>call 0x20040040
+Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.0.210/255.255.255.0, Gateway: 10.0.0.1
Default server: 10.0.0.102, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version v2_0_11a1 - built 13:21:01, Feb 12 2004

Platform: Cogent CSB337 (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x02000000, [0x00065e40-0x01fdd000] available
FLASH: 0x60000000 - 0x60800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

Now the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m xmodem
```

Again, use HyperTerminal's Xmodem support to send the file `redboot_ROMRAM.bin`. This should result in something like the following output:

```
Raw file loaded 0x00030000-0x0004d15f, assumed entry at 0x00030000
xyzModem - CRC mode, 932(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROMRAM.bin
2299507324 119136 redboot_ROMRAM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 119136
```

```
POSIX cksum = 2299507324 119136 (0x890fb27c 0x0001d160)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the **-l** option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init -f
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x60020000-0x607e0000: .....
... Erase from 0x60800000-0x60800000:
... Unlock from 0x607e0000-0x60800000: .
... Erase from 0x607e0000-0x60800000: .
... Program from 0x01fe0000-0x02000000 at 0x607e0000: .
... Lock from 0x607e0000-0x60800000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x60000000-0x60020000: .
... Program from 0x00100000-0x00120000 at 0x60000000: .
... Unlock from 0x607e0000-0x60800000: .
... Erase from 0x607e0000-0x60800000: .
... Program from 0x01fe0000-0x02000000 at 0x607e0000: .
... Lock from 0x607e0000-0x60800000: .
RedBoot>
```

The CBS337 board may now be reset either by cycling the power, pressing the reset switch, or with the **reset** command. It should then display the startup screen for the ROMRAM version of RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the CSB337 are:

```
$ mkdir redboot_csb337_romram
$ cd redboot_csb337_romram
$ ecosconfig new csb337 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/csb337/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the RAM version of RedBoot:

```
$ mkdir redboot_csb337_ram
$ cd redboot_csb337_ram
$ ecosconfig new csb337 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/csb337/VERSION/misc/redboot_RAM.ecm
$ ecosconfig resolve
```

Setup

```
$ ecosconfig tree  
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the RAM RedBoot into the ROM will render the board unbootable.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The CSB337 platform HAL package is loaded automatically when eCos is configured for a `csb337` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The CSB337 platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x10000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location `0x10000000`. However, when it starts up the application will first copy itself to RAM at `0x00000000` and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The CBS337 board contains an 8Mb Intel StrataFlash flash device. The `CYGPKG_DEVS_FLASH_STRATA` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_CSB337` package contains definitions that customize the driver to the CSB337 board.

Ethernet Driver

The CSB337 board uses the AT91RM9200's internal EMAC ethernet device attached to an external Intel LXT971 PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_CSB337` package contains definitions that customize the driver to the CSB337 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

```
-mcpu=arm9
```

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the CSB337 hardware, and should be read in conjunction with that specification. The CSB337 platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. However, the HAL uses the MMU to relocate it to virtual address 0x60000000 after initialization.

SDRAM

This is located at address 0x20000000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x20000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

Apart from the SDRAM, flash and ethernet PHY, eCos does not currently make any use of the off-chip peripherals present on the CSB337.

Other Issues

The CSB337 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXII. KwikByte KB920x Board Family Support

Overview

Name

eCos Support for the KB920x Board Family — Overview

Description

This document covers the KwikByte KB920x family of single board computers based on the Atmel AT91RM9200. The KB9200 contains the AT91RM9200 processor, 32Mb of SDRAM, 2MB of flash memory, an Intel LXT971 PHY and external connections for one serial channel, ethernet and the various other peripherals supported by the AT91RM9200. The KB9201 is similar but with an additional SPI dataflash. The KB9202 uses a different flash memory device and increases the flash memory capacity to 16MB. The KB9202B extends SDRAM to 64MB, and the KB9202C replaces the NOR flash with Dataflash.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

On the KB9200 and KB9201, the flash memory consists of 32 blocks of 64k bytes each. On the KB9202, the flash memory consists of 128 blocks of 128k bytes each. In a typical setup, the first two flash blocks are used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. For the KB9200/KB9201 these are 29 blocks between 0x60020000 and 0x601EFFFF; for the KB9202 these are 125 blocks between 0x60020000 and 0x60FDFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the KB9200, KB9201 or KB9202 targets.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_KB9200` is responsible for configuring this generic driver to the KB920x hardware. These drivers are also loaded automatically when configuring for the KB9200, KB9201 or KB9202 targets.

eCos manages the on-chip interrupt controller. Timer counter 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded.

Tools

The KB920x port is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14, and subsequently retested with arm-elf-gcc version 3.4.3, arm-elf-gdb version 6.3 and binutils version 2.16.

Setup

Name

Setup — Preparing the KB920x boards for eCos Development

Overview

In a typical development environment, the KB920x boards boot from serial EEPROM into the KwikByte bootloader, which then boots the RedBoot ROM monitor from flash. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.bin
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector.	redboot_ROMRAM.ecm	redboot_ROMRAM.bin
SRAM	RedBoot running from RAM, loaded by bootloader.	redboot_SRAM.ecm	redboot_SRAM.bin
UBOOT	RedBoot running from RAM, loaded from flash by U-Boot.	redboot_UBOOT.ecm	redboot_UBOOT.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

Generally only the ROMRAM version of RedBoot is used. The SRAM version is suitable only for RedBoot development and should not normally be used. On the KB9202C the UBOOT version is used.

Initial Installation -- KB9201, KB9202, KB9202B

The KB920x boards are shipped from KwikByte with the EEPROM bootloader installed and a version of Linux installed in the flash. The KB9202C board has a different installation method, described in the next section.

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Linux image. This is a two stage process, you must first download the KwikByte RAM Monitor, which is then used to download a RedBoot image and program it into the flash. To achieve this you will need a copy of the `ramMonitor.bin` image and, for Linux hosts, a copy of the KwikByte **download** tool. Both of these can be found on the CD-ROM that comes with the board. Note that the prebuilt version of the **download** tool is hardcoded to use `/dev/ttyS0` for downloads.

Connect a straight-through serial cable between the KB920x board serial port and a serial port of the host computer

and start a terminal emulator such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit. Press the Reset button on the board and when the bootloader starts you should see something similar to this if using the KB9200 or KB9201:

```
KB9200(www.kwikbyte.com)
    Default system configuration complete
```

```
Checking for input
```

```
>
```

When you see the "Checking for input" line, hit the Return key to put the bootloader into its command interpreter, otherwise it will proceed to boot Linux.

For the KB9202 there is less output, and the delay shorter, and so you will need to press the Return key several times immediately after pressing the Reset button before the bootloader banner is displayed:

```
KB9202(www.kwikbyte.com)
```

```
Auto boot..
```

```
>
```

The next step is to install a new startup command in the command table:

```
>s 1 e 0x10000000
```

```
>w
```

```
>
```

It is now necessary to download the Kwikbyte RAM Monitor. Start the download by giving the following command to the bootloader:

```
>x 0x20002000
C
```

You may get a sequence of C characters, which indicate that the bootloader is waiting for the download to start. If using HyperTerminal, you should now send the `ramMonitor.bin` file from the KwikByte CD-ROM using the Xmodem protocol. If using **minicom**, exit **minicom** and download the RAM Monitor using the following command:

```
$ download ramMonitor.bin
```

The **download** command produces a lot of output. When the transfer is finished, restart **minicom**.

Having downloaded the RAM monitor, issue the following command to start it:

```
>e 0x20002000
```

```
Entry: RAM Monitor
```

```
>
```

It is now necessary to prepare the flash for the RedBoot image by erasing the first 128K bytes. The RedBoot ROM image can then be downloaded:

```
>f e 0x10000000 0x1001ffff
Verifying sector erase
Flash sector erase at: 0x10000000 PASS
Verifying sector erase
Flash sector erase at: 0x10010000 PASS
```

```
>x 0x20100000
C
```

If using HyperTerminal, you should now send the `redboot_ROMRAM.bin` file using the Xmodem protocol. If using **minicom**, exit **minicom** and use the **download** command to send the file `redboot_ROMRAM.bin`. When the transfer is finished, restart **minicom**.

Having downloaded the RedBoot image, use the following command to write it to flash:

```
>f p 0x10000000 0x20100000 0x20000
..
Flash program status: PASS

>
```

The KB920x board may now be reset. It should then display the following startup sequence. Run the **fis init** and **fconfig -i** commands to initialize the flash, as shown. Note that KB9202 users will see output similar but not identical to that below.

```
KB9200(www.kwikbyte.com)
    Default system configuration complete

Checking for input
0x00 : [E]
0x01 : e 0x10000000[E]
0x02 : c 0x20210000 0x10100000 0x100000[E]
0x03 : m 0 0 0 0 0 0[E]
0x04 : t 0x20000100 console=ttyS0,115200 root=/dev/ram rw initrd=0x20210000,654]
0x05 : e 0x10000000[E]
0x06 : [E]
0x07 : [E]
0x08 : [E]
0x09 : [E]

>

>e 0x10000000
+... Read from 0x601f0000-0x601fffff to 0x01ff0000:
... Read from 0x601ff000-0x601fffff to 0x01fef000:
**Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
PHY: Intel LXT971
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb
```

```
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 12:01:32, Dec 13 2004

Platform: KwikByte KB9200 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x0002ddf8-0x01fed000] available
FLASH: 0x60000000 - 0x601fffff 32 x 0x10000 blocks
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Erase from 0x601f0000-0x601fffff: .
... Program from 0x01ff0000-0x02000000 to 0x601f0000: .
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: true
Default server IP address: 10.0.0.201
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x1C
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91rm9200_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Read from 0x601f0000-0x601fefff to 0x01ff0000:
... Erase from 0x601f0000-0x601fffff: .
... Program from 0x01ff0000-0x02000000 to 0x601f0000: .
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Initial Installation -- KB9202C

The KB9202C is not equipped with NOR flash, only Dataflash, so the installation method described above will not work. This board boots, via a second-stage loader into U-Boot. By default U-Boot then boots a Linux image that is also stored in Dataflash. The approach described here is to cause U-Boot to boot RedBoot instead of Linux. The following directions give basic instructions for installing RedBoot, for more details of how to configure U-Boot, please refer to the U-Boot documentation.

Connect a straight-through serial cable between the KB920C board serial port and a serial port of the host computer and start a terminal emulator such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit. Power the board up and you should see the KwikByte loader and U-Boot startup messages. When it gets to the "Hit any key to stop autoboot" line, type any key. There is only a 1

second timeout here, so you need to be quick, or you can type ahead during the delay after the "NAND:" line. The final output should look like this:

```
KwikByte KB9202x Copy Loader v0.9
Loading boot loader. . . done
```

```
U-Boot 1.2.0 (Sep 26 2007 - 17:32:22)
```

```
DRAM: 64 MB
NAND: NAND device: Manufacturer ID: 0x2c, Chip ID: 0xda (Micron NAND 256MiB 3,3V 8-bit)
256 MiB
DataFlash:AT45DB642
Nb pages: 8192
Page Size: 1056
Size= 8650752 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C00020FF Copy Loader (8kB)
Area 1: C0002100 to C00041FF Environment (8kB)
Area 2: C0004200 to C003665F U-Boot (195kB)
Area 3: C0036660 to C0041FFF Secondary Boot (45kB)
Area 4: C0042000 to C0461FFF OS (4MB)
Area 5: C0462000 to C083FFFF DF_SPARE
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
KB920x >
```

Now we need to set up U-Boot to download the RedBoot image and program it into the DataFlash. First it is necessary to change U-Boot's environment to allow the image to be downloaded. In this example we will be downloading via tftp, so we need to set the ethernet and IP addresses for this board:

```
KB920x >setenv ethaddr 00:D0:93:00:05:B5
KB920x >setenv ipaddr 10.0.3.1
KB920x >setenv serverip 10.0.1.2
KB920x >saveenv
KB920x >printenv
bootargs=console=ttyS0,115200 noinitrd root=/dev/mtdblock0 rootfstype=jffs2 mem=
64M
bootdelay=1
baudrate=115200
hostname=KB9202C
kernel-size=31c8d0
bootcmd=cp.b c0042000 23000000 31c8d0; bootm 23000000
ethaddr=00:D0:93:00:05:B5
ipaddr=10.0.3.1
serverip=10.0.1.2
stdin=serial
stdout=serial
stderr=serial

Environment size: 307/8444 bytes
KB920x >
```

In the above example the `ipaddr` and `serverip` variables are set to example values. They should be set to the address of this board and the address of the TFTP server in your own network. The `redboot_UBOOT.bin` should be copied to the tftp server's root directory. It may now be downloaded:

```
KB920x >tftp 0x20100000 redboot_UBOOT.bin
TFTP from server 10.0.1.2; our IP address is 10.0.3.1
Filename 'redboot_UBOOT.bin'.
Load address: 0x20100000
Loading: T #####
done
Bytes transferred = 94236 (1701c hex)
KB920x >
```

Now program the loaded binary into flash. In the following example, the new binary is installed into dataflash at offset 0x500000. This avoids overwriting the Linux image at offset 0x42000. If the Linux image is not required, RedBoot can be programmed in its place and the addresses in the following commands adjusted to match.

```
KB920x >cp.b 0x20100000 0xc0500000 0x20000
Copy to DataFlash... done
KB920x >
```

Finally, change the default boot command to load and execute RedBoot rather than Linux. Make sure to include the backslash before the semicolon.

```
KB920x >setenv bootcmd cp.b 0xc0500000 0x20100000 0x20000\; go 0x20100000
KB920x >saveenv
Saving Environment to dataflash...
```

Press the reset button on the board and the full boot sequence should be seen:

```
KwikByte KB9202x Copy Loader v0.9
Loading boot loader. . . done
```

```
U-Boot 1.2.0 (Sep 26 2007 - 17:32:22)
```

```
DRAM: 64 MB
NAND: NAND device: Manufacturer ID: 0x2c, Chip ID: 0xda (Micron NAND 256MiB 3,3V 8-bit)
256 MiB
DataFlash:AT45DB642
Nb pages: 8192
Page Size: 1056
Size= 8650752 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C00020FF Copy Loader (8kB)
Area 1: C0002100 to C00041FF Environment (8kB)
Area 2: C0004200 to C003665F U-Boot (195kB)
Area 3: C0036660 to C0041FFF Secondary Boot (45kB)
Area 4: C0042000 to C0461FFF OS (4MB)
Area 5: C0462000 to C083FFFF DF_SPARE
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
```

```
.O+AT91RM9200_ETH - Warning! ESA unknown.
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.6/255.0.0.0, Gateway: 10.0.0.3
Default server: 0.0.0.0, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [UBOOT]
Non-certified release, version UNKNOWN - built 17:30:00, Jan 11 2008

Platform: KwikByte KB9202C (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006, 2007 eCosCentric Limited

RAM: 0x00000000-0x04000000, [0x00120840-0x04000000] available
RedBoot>
```

If it is necessary to reinstall RedBoot, the above steps for downloading and programming the new image should be repeated.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the KB9200 are:

```
$ mkdir redboot_kb9200_romram
$ cd redboot_kb9200_romram
$ ecosconfig new kb9200 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/kb9200/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the the ROMRAM version of RedBoot for the KB9202 are:

```
$ mkdir redboot_kb9202_romram
$ cd redboot_kb9202_romram
$ ecosconfig new kb9202 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/kb9200/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

The steps needed to rebuild the the UBOOT version of RedBoot for the KB9202C are:

```
$ mkdir redboot_kb9202c_uboot
$ cd redboot_kb9202c_uboot
$ ecosconfig new kb9202c redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/kb9200/VERSION/misc/redboot_UBOOT.ecm
$ ecosconfig resolve
$ ecosconfig tree
```

Setup

```
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The KB920x platform HAL package is loaded automatically when eCos is configured for a kb9200 or kb9202 targets. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The KB920x platform HAL package supports four separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. arm-elf-gdb is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address 0x10000000. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location 0x10000000. However, when it starts up the application will first copy itself to RAM at 0x00000000 and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

SRAM

This startup type is used for applications that are downloaded via the KwikByte bootloader directly to RAM. The application is loaded into SDRAM at location 0x20000000 and started by executing from that address. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform limited hardware initialization since it is assumed that the machine has been set up already by the bootloader.

This configuration is primarily present as a result of the development process. It has some limitations with regard to functionality since the MMU is not enabled and no exception vectors are installed at location zero, hence no interrupts can be handled.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The KB9200 and KB9201 boards contain a 2Mb AMD Am29LV017D flash device. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX` package contains all the code necessary to support this part and the `CYGPKG_DEVS_FLASH_KB9200` package contains definitions that customize the driver to the KB9200 board.

The KB9202 board contains a 16Mb Intel StrataFLASH 28F128J3 flash device. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support this part and the platform HAL contains definitions that customize the driver to the KB9202 board.

Ethernet Driver

The KB920x boards use the AT91RM9200's internal EMAC ethernet device attached to an external Intel LXT971 PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_KB9200` package contains definitions that customize the driver to the KB920x boards.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=arm9`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the KB920x hardware, and should be read in conjunction with that specification. The KB920x platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. The HAL uses the MMU to locate it at virtual address 0x60000000 after initialization. It remains accessible at address 0x10000000 but accesses to this address range are uncached.

SDRAM

This is located at address 0x20000000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x20000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered

at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

Apart from the SDRAM, flash and ethernet PHY, eCos does not currently make any use of the off-chip peripherals present on the KB920x boards.

Real-time characterization

The tm_basic kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in Thumb mode, which provided better performance than ARM mode.

```
Startup, main stack : stack used    336 size  3920
Startup : Interrupt stack used    492 size  4096
Startup : Idlethread stack used     88 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

```
Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took  15.40 microseconds (7 raw clock ticks)
```

Testing parameters:

```
Clock samples:      32
Threads:            64
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32
```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
16.70	10.67	23.47	3.01	51%	28%	Create thread
2.50	2.13	8.53	0.63	85%	85%	Yield thread [all suspended]
2.67	2.13	6.40	0.82	76%	76%	Suspend [suspended] thread
3.00	2.13	6.40	1.06	60%	60%	Resume thread
4.10	2.13	10.67	0.49	85%	12%	Set priority
1.23	0.00	2.13	1.04	57%	42%	Get priority
8.30	6.40	21.34	0.77	78%	20%	Kill [suspended] thread

2.73	2.13	8.53	0.90	75%	75% Yield [no other] thread
4.50	4.27	8.53	0.42	90%	90% Resume [suspended low prio] thread
2.70	2.13	6.40	0.85	75%	75% Resume [runnable low prio] thread
4.33	2.13	8.53	0.32	89%	4% Suspend [runnable] thread
2.70	2.13	4.27	0.83	73%	73% Yield [only low prio] thread
2.77	2.13	4.27	0.89	70%	70% Suspend [runnable->not runnable]
8.20	6.40	21.34	0.90	73%	25% Kill [runnable] thread
7.40	6.40	14.93	1.15	98%	57% Destroy [dead] thread
11.73	10.67	25.60	1.27	98%	59% Destroy [runnable] thread
14.67	12.80	25.60	0.87	73%	23% Resume [high priority] thread
6.72	6.40	10.67	0.54	85%	85% Thread switch
0.55	0.00	2.13	0.82	74%	74% Scheduler lock
1.80	0.00	2.13	0.56	84%	15% Scheduler unlock [0 threads]
1.82	0.00	2.13	0.54	85%	14% Scheduler unlock [1 suspended]
1.82	0.00	4.27	0.57	83%	15% Scheduler unlock [many suspended]
1.82	0.00	4.27	0.57	83%	15% Scheduler unlock [many low prio]
0.87	0.00	2.13	1.03	59%	59% Init mutex
3.13	2.13	6.40	1.12	56%	56% Lock [unlocked] mutex
3.53	2.13	6.40	1.05	59%	37% Unlock [locked] mutex
3.13	2.13	6.40	1.12	56%	56% Trylock [unlocked] mutex
2.60	2.13	6.40	0.76	81%	81% Trylock [locked] mutex
0.87	0.00	2.13	1.03	59%	59% Destroy mutex
15.13	14.93	21.34	0.38	96%	96% Unlock/Lock mutex
1.27	0.00	4.27	1.11	53%	43% Create mbox
0.93	0.00	2.13	1.05	56%	56% Peek [empty] mbox
3.40	2.13	6.40	1.11	53%	43% Put [first] mbox
0.87	0.00	2.13	1.03	59%	59% Peek [1 msg] mbox
3.27	2.13	6.40	1.13	96%	50% Put [second] mbox
1.07	0.00	2.13	1.07	100%	50% Peek [2 msgs] mbox
3.40	2.13	6.40	1.11	53%	43% Get [first] mbox
3.73	2.13	6.40	0.90	68%	28% Get [second] mbox
3.53	2.13	6.40	1.05	59%	37% Tryput [first] mbox
3.53	2.13	8.53	1.14	56%	40% Peek item [non-empty] mbox
3.53	2.13	6.40	1.05	59%	37% Tryget [non-empty] mbox
3.33	2.13	6.40	1.12	50%	46% Peek item [empty] mbox
3.33	2.13	6.40	1.12	50%	46% Tryget [empty] mbox
1.20	0.00	2.13	1.05	56%	43% Waiting to get mbox
1.20	0.00	4.27	1.12	50%	46% Waiting to put mbox
3.53	2.13	8.53	1.14	56%	40% Delete mbox
7.60	6.40	17.07	1.42	93%	59% Put/Get mbox
0.87	0.00	2.13	1.03	59%	59% Init semaphore
2.33	2.13	4.27	0.36	90%	90% Post [0] semaphore
2.87	2.13	6.40	1.01	68%	68% Wait [1] semaphore
2.60	2.13	6.40	0.76	81%	81% Trywait [0] semaphore
2.27	2.13	4.27	0.25	93%	93% Trywait [1] semaphore
0.73	0.00	2.13	0.96	65%	65% Peek semaphore
1.00	0.00	4.27	1.12	56%	56% Destroy semaphore
9.07	8.53	17.07	0.90	84%	84% Post/Wait semaphore
1.20	0.00	4.27	1.12	50%	46% Create counter
0.87	0.00	2.13	1.03	59%	59% Get counter value

0.60	0.00	2.13	0.86	71%	71%	Set counter value
3.33	2.13	6.40	1.12	50%	46%	Tick counter
1.27	0.00	2.13	1.03	59%	40%	Delete counter
0.73	0.00	2.13	0.96	65%	65%	Init flag
2.60	2.13	6.40	0.76	81%	81%	Destroy flag
2.47	2.13	4.27	0.56	84%	84%	Mask bits in flag
2.80	2.13	6.40	0.96	71%	71%	Set bits in flag [no waiters]
4.00	2.13	8.53	0.70	78%	18%	Wait for flag [AND]
3.67	2.13	8.53	1.05	62%	34%	Wait for flag [OR]
3.67	2.13	8.53	1.05	62%	34%	Wait for flag [AND/CLR]
3.67	2.13	8.53	1.05	62%	34%	Wait for flag [OR/CLR]
0.40	0.00	2.13	0.65	81%	81%	Peek on flag
0.67	0.00	4.27	0.96	71%	71%	Create alarm
4.67	4.27	10.67	0.70	87%	87%	Initialize alarm
2.73	2.13	6.40	0.90	75%	75%	Disable alarm
4.60	4.27	10.67	0.60	90%	90%	Enable alarm
3.13	2.13	6.40	1.12	56%	56%	Delete alarm
3.07	2.13	4.27	1.05	56%	56%	Tick counter [1 alarm]
16.60	14.93	17.07	0.73	78%	21%	Tick counter [many alarms]
5.07	4.27	8.53	1.05	65%	65%	Tick & fire counter [1 alarm]
87.67	87.48	89.61	0.36	90%	90%	Tick & fire counters [>1 together]
18.80	17.07	21.34	0.76	75%	21%	Tick & fire counters [>1 separately]
12.97	12.80	32.00	0.33	98%	98%	Alarm latency [0 threads]
15.03	12.80	36.27	0.97	65%	18%	Alarm latency [2 threads]
15.28	12.80	40.54	1.64	39%	27%	Alarm latency [many threads]
24.09	23.47	98.14	1.20	97%	97%	Alarm -> thread resume latency
2.27	2.13	10.67	0.00			Clock/interrupt latency
5.40	4.27	12.80	0.00			Clock DSR latency
10	0	764	(main stack: 1328) Thread stack used (1360 total)			
			All done, main stack : stack used 1328 size 3920			
			All done : Interrupt stack used 136 size 4096			
			All done : Idlethread stack used 212 size 2048			

Timing complete - 29880 ms total

PASS:<Basic timing OK>

EXIT:<done>

Other Issues

The KB920x platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXIII. SSV DNP/9200 with DNP/EVA9 Board Support

Overview

Name

eCos Support for the SSV DNP/9200 with DNP/EVA9 Evaluation Board — Overview

Description

This document covers the configuration and usage of eCos and RedBoot on the DNP/9200 with DNP/EVA9 evaluation board as provided in the SSV SK23 starter kit. This kit provides the AT91RM9200 processor, 32Mbytes of SDRAM, 16Mbytes of parallel NOR flash memory, a Davicom DM9161A PHY, a SD/MMC/DataFlash socket, a DAC, external connections for two serial channels, ethernet, USB host/device and the various other peripherals supported by the AT91RM9200. eCos support for the many devices and peripherals on the boards and the AT91RM9200 is described below.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot into this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

This documentation is expected to be read in conjunction with the AT91RM9200 processor HAL documentation and further device support and subsystems are described and documented there.

Supported Hardware

The parallel NOR flash memory consists of 128 blocks of 128Kbytes each. In a typical setup, the first 256 Kbytes are reserved for the use of the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining blocks can be used by application code. There are 126 blocks available between 0x60030000 and 0x60FDFFFF.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_AT91` which supports both the Debug Unit and USART serial devices. The serial port at J6 COM1 (connected to USART channel 1) and DTE port at J7 COM2 (connected to USART channel 2) can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the DNP/9200 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_ARM_AT91RM9200` for the on-chip ethernet device. The platform HAL package is responsible for configuring this generic driver to the hardware. This driver is also loaded automatically when configuring for the DNP/9200 target.

There is a watchdog driver `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200`. This driver is also loaded automatically when configuring for the DNP/9200 target.

There is a driver for the on-chip real-time clock (RTC) at `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91`. This driver is also loaded automatically when configuring for the DNP/9200 target.

The AT91RM9200 processor HAL contains a driver for the Two-Wire Interface (TWI) controller on the AT91RM9200. This type of bus is also known as I²C®. Further documentation may be found in the AT91RM9200 processor HAL documentation.

The AT91RM9200 processor HAL contains a driver for the MultiMedia Card Interface (MCI). This driver is loaded automatically when configuring for the DNP/9200 target and allows use of MMC and Secure Digital (SD) flash storage cards within eCos, exported as block devices. Further documentation may be found in the AT91RM9200 processor HAL documentation.

There is a general AT91 SPI driver (`CYGPKG_DEVS_SPI_ARM_AT91`) which provides the underlying implementation for the SPI API layer in the `CYGPKG_IO_SPI` package. All these packages are automatically loaded when configuring for the DNP/9200 target.

In general, devices (Caches, PIO, UARTs, EMAC) are initialized only as far as is necessary for eCos to run. Other devices (RTC, I2C, SPI, MCI etc.) are not touched unless the appropriate driver is loaded, although in some cases, the HAL boot sequence will set up the appropriate PIO configuration.

Tools

The DNP/9200 support is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.4.4, arm-elf-gdb version 6.3, and binutils version 2.15.

Setup

Name

Setup — Preparing the DNP/9200 with DNP/EVA9 evaluation board for eCos Development

Overview

In a typical development environment, the DNP/9200 board boots from the parallel NOR Flash and run the RedBoot ROM monitor directly. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.bin
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.bin
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	redboot_ROMRAM.ecm	redboot_ROMRAM.bin
JTAG	RedBoot running from RAM, loaded via JTAG	redboot_JTAG.ecm	redboot_JTAG.bin
UBOOT	RedBoot running from RAM, loaded via U-Boot	redboot_UBOOT.ecm	redboot_UBOOT.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 115200 baud. RedBoot also supports ethernet communication and flash management.

The ample provision of RAM memory on the board allows the ROMRAM version of RedBoot to be preferential to the standard ROM version which executes directly from Flash. Alternatively, if the ROM version is to be chosen, then the RAM version is provided to allow for updating the resident RedBoot image in Flash. The JTAG version is only used if loading RedBoot into RAM via a JTAG debugger or ICE. It is similar to the RAM version, but loads at a lower address within RAM, and so can be used to in turn load eCos applications, as if it is the normal resident boot monitor. The ELF format image of this JTAG version of RedBoot can also be loaded and executed from GDB using the Abatron BDI2000 bdiGDB support, to allow it to be debugged. The UBOOT version may be used to load RedBoot into RAM using the UBOOT bootloader. This is usually only used to then load a ROMRAM executable for programming into flash. Like the JTAG version it loads at a lower RAM address and can therefore be used to load RAM applications.

Initial Installation

The on-chip boot program on the AT91RM9200 is only capable of loading programs into 12Kbytes of on-chip SRAM and is therefore quite restrictive. Consequently two mechanisms are described below to program RedBoot

into Flash. The first requires a JTAG device while the other makes use of the U-Boot bootloader that is shipped with the board.

Method 1 - Program RedBoot into Flash with RAM RedBoot loaded by JTAG

With this approach, the BDI2000 is used to load a RAM RedBoot image, which can then in turn be used to load and program a ROMRAM RedBoot image into Flash. In the following documentation it is assumed that the Abatron BDI2000 is being used. For a different JTAG device, equivalent operations will need to be performed.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot. First, however, we must set up the BDI2000 and the board.

Preparing the Abatron BDI2000 JTAG debugger

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Install the Abatron BDI2000 bdiGDB support software on the host PC.
4. Locate the file `bdi2000.dnp_sk23.cfg` within the eCos platform HAL package in the source repository. This will be in the directory `packages/hal/arm/arm9/dnp_sk23/VERSION/misc` relative to the root of your eCos installation.
5. Locate the file `reg920t.def` within the installation of the BDI2000 bdiGDB support software.
6. Place the `bdi2000.dnp_sk23.cfg` in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file.
7. Similarly place the file `reg920t.def` in a location accessible to the TFTP server.
8. Open `bdi2000.dnp_sk23.cfg` in an editor such as emacs or notepad and if necessary adjust the path of the `reg920t.def` file in the `[REGS]` section to match its location relative to the TFTP server root.
9. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the `bdi2000.dnp_sk23.cfg` configuration file at the appropriate point of this process.

Preparing the DNP/9200 with DNP/EVA9 evaluation board for programming

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. First you must connect a null modem DB9 serial cable between COM1 on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to your host PC's LAN with an Ethernet cable.

4. You should designate the board with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.
5. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG interface connector on the DNP/9200 to the Target A port on the BDI2000. Since the JTAG connector on the DNP/9200 is non-standard, this will require an adaptor cable.
6. Power up the DNP/EVA9 board. You should see the power LED and some of the ethernet LEDs illuminate.
7. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
Core#0>
```

8. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
Core#0> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x00000001 => 0x00000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x05B0203F
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

9. Locate the `redboot_ROMRAM.bin` image within the `loaders` subdirectory of the base of the eCos installation.
10. Copy the `redboot_ROMRAM.bin` file into a location on the host computer accessible to its TFTP server.

Loading a RAM RedBoot

1. Locate the `redboot_JTAG.bin` image within the `loaders` subdirectory of the base of the eCos installation.
2. Copy the `redboot_JTAG.bin` file into a location on the host computer accessible to its TFTP server.
3. With the BDI2000 telnet interface, execute the following command, replacing `/RBPATH` with the location of the `redboot_JTAG.bin` file relative to the TFTP server root directory:

```
Core#0>load 0x20010000 /RBPATH/redboot_JTAG.bin bin
Loading /RBPATH/redboot_JTAG.bin , please wait ....
Loading program file passed
Core#0>
```

4. Run the loaded RAM RedBoot:

```
Core#0>go 0x20010040
Core#0>
```

The terminal emulator connected to the serial debug port should now have displayed the RedBoot banner and prompt similar to the following:

```
***Warning** FLASH configuration checksum error or invalid key
Use 'fconfig -i' to [re]initialize database
PHY: Davicom DM9161A
AT91RM9200 ETH: Waiting for link to come up.
AT91RM9200 ETH: 100Mb
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version UNKNOWN - built 17:10:26, Jun  8 2006

Platform: DNP/9200 with DNP/EVA9 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x20000000-0x20800000, [0x2002f2e0-0x207dd000] available
FLASH: 0x60000000-0x60ffffff, 128 x 0x20000 blocks
RedBoot>
```

In the above output, a local BOOTP/DHCP server was able to serve an address to the device.

Note: It is also possible to use the RAM startup version of RedBoot and the redboot_RAM.bin file instead of redboot_JTAG.bin above. If so, then the address to the **load** command must be 0x20100000, as must be the address to the **go** command.

RedBoot Flash configuration

The following steps describe how to initialize RedBoot's Flash configuration. This must be performed when using a RAM RedBoot to program Flash.

1. Use the following command to initialize RedBoot's Flash Information System (FIS):

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)? y
*** Initialize FLASH Image System
... Unlocking from 0x60fe0000-0x60ffffff: .
... Erase from 0x60fe0000-0x60ffffff: .
... Program from 0x207e0000-0x20800000 to 0x60fe0000: .
... Locking from 0x60fe0000-0x60ffffff: .
RedBoot>
```

2. Now configure RedBoot's Flash configuration with the command:

```
RedBoot> fconfig -i
```


Remember to substitute the appropriate MAC address for this board at the appropriate step. If a BOOTP/DHCP server is not available, then IP configuration may be set manually. The default server IP address can be set to a PC that will act as a TFTP host for future RedBoot load operations, or may be left unset. The following gives an example configuration:

```
RedBoot> fconfig -i
Initialize non-volatile configuration - continue (y/n)? y
Run script at boot: false
Use BOOTP for network configuration: false
Gateway IP address: 192.168.7.11
Local IP address: 192.168.7.222
Local IP address mask: 255.255.255.0
Default server IP address: 192.168.7.9
Console baud rate: 115200
DNS server IP address: 192.168.7.11
Network hardware address [MAC]: 0x00:0x23:0x31:0x37:0x00:0x4e
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Default network device: at91rm9200_eth
Update RedBoot non-volatile configuration - continue (y/n)? y
... Unlocking from 0x60fe0000-0x60ffffff: .
... Erase from 0x60fe0000-0x60ffffff: .
... Program from 0x207e0000-0x20800000 to 0x60fe0000: .
... Locking from 0x60fe0000-0x60ffffff: .
RedBoot>
```

Loading and programming the ROMRAM RedBoot

This section describes the steps required to load the ROMRAM RedBoot from the TFTP server and program it into Flash.

1. Load the RedBoot ROMRAM binary image from the TFTP server. Use the following command, replacing *111.222.333.444* with the TFTP server IP address (or domain name if a DNS server has been configured), and */RBPATH* with the location of the *redboot_ROMRAM.bin* file relative to the TFTP server root directory:

```
RedBoot> load -r -b {%freememlo} -h 111.222.333.444 /RBPATH/redboot_ROMRAM.bin
Using default protocol (TFTP)
Raw file loaded 0x20030000-0x2004e91b, assumed entry at 0x20030000
RedBoot>
```

2. Finally install the loaded image into Flash:

```
RedBoot> fis create RedBoot
An image named 'RedBoot' exists - continue (y/n)? y
... Erase from 0x60000000-0x6003ffff: ..
... Program from 0x20030000-0x2004e91c to 0x60000000: .
... Locking from 0x60000000-0x6003ffff: ..
... Unlocking from 0x60fe0000-0x60ffffff: .
... Erase from 0x60fe0000-0x60ffffff: .
```

```
... Program from 0x21fe0000-0x22000000 to 0x60fe0000: .  
... Locking from 0x60fe0000-0x60ffffff: .  
RedBoot>
```

It is also possible to use the **fis write** command to write the image into Flash, but if so, the relevant Flash blocks must also be explicitly unlocked with the command:

```
RedBoot> fis unlock -f 0x60000000 -l 0x30000
```

The RedBoot installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. Output similar to the following should be seen on the serial port. Verify the IP settings are as expected.

```
+PHY: Davicom DM9161A  
AT91RM9200 ETH: 100Mb  
... waiting for BOOTP information  
Ethernet eth0: MAC address 00:23:31:37:00:1c  
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3  
Default server: 10.0.1.2, DNS server IP: 10.0.0.1  
  
RedBoot(tm) bootstrap and debug environment [ROMRAM]  
Non-certified release, version UNKNOWN - built 18:10:00, Jun  8 2006  
  
Platform: DNP/9200 with DNP/EVA9 (ARM9)  
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.  
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited  
  
RAM: 0x20000000-0x22000000, [0x2002ff78-0x21fdd000] available  
FLASH: 0x60000000-0x60ffffff, 128 x 0x20000 blocks  
RedBoot>
```

If it proves necessary to re-install RedBoot, this may be achieved by repeating the above process. Alternatively, a new image may be downloaded and programmed into flash more directly using RedBoot's own commands. See the RedBoot documentation for details.

Method 2 - Program RedBoot into Flash with RAM Redboot loaded by U-Boot

With this approach, the existing U-Boot bootloader is used to load a RAM RedBoot which is then used to load and program a ROMRAM RedBoot image to replace U-Boot in Flash. A JTAG debugger is not needed for this method.

There are three stages, firstly loading the RAM RedBoot image, then initializing RedBoot's Flash configuration, and finally loading and programming the ROMRAM RedBoot. The first of these stages is described here, the remaining two stages are identical to the equivalent stages in Method 1, above.

Loading a RAM RedBoot

1. First you must connect a null modem DB9 serial cable between COM1 on the board and a serial port on the host computer.
2. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
3. Connect the board to your host PC's LAN with an Ethernet cable.
4. You should designate the board with a new Ethernet MAC address. The RedBoot binary image contains a default address, but each board requires its own unique address. It is advisable to mark each board with its programmed MAC address for future identification.
5. Locate the `redboot_UBOOT.bin` image within the `loaders` subdirectory of the base of the eCos installation.
6. Copy the `redboot_UBOOT.bin` file into a location on the host computer accessible to its TFTP server.
7. Power up the DNP/EVA9 board. You should see the power LED and some of the ethernet LEDs illuminate.
8. After a few seconds the following output should be seen on the serial line. Hit a key to stop the auto boot:

```

U-Boot 1.1.2 (Dec 14 2005 - 13:12:14)

U-Boot code: 21F00000 -> 21F1666C BSS: -> 21F1AC44
RAM Configuration:
Bank #0: 20000000 32 MB
Flash: 16 MB

In:      serial
Out:     serial
Err:     serial
Hit any key to stop autoboot:  0
U-Boot>

```

9. Set up the environment of U-Boot to download the RAM RedBoot. Users with access to a TFTP server should substitute their own IP address, TFTP server address and netmask in the following commands:

```

U-Boot> setenv ipaddr 10.0.2.22
U-Boot> setenv serverip 10.0.1.2
U-Boot> setenv netmask 255.0.0.0
U-Boot> setenv bootfile redboot_UBOOT.bin

```

10. Download the RAM RedBoot. Users with access to a TFTP server should use the following command:

```

U-Boot> tftpboot 0x20040000
TFTP from server 10.0.1.2; our IP address is 10.0.2.22
Filename 'redboot_UBOOT.bin'.
Load address: 0x20040000
Loading: #####
done
Bytes transferred = 120760 (1d7b8 hex)

```

11. Users without access to a TFTP server may use Kermit to send the file `redboot_UBOOT.bin`:

```

U-Boot> loadb 0x20040000

```

12. The downloaded RedBoot image may now be executed:

```
U-Boot> go 0x20040000
.+PHY: Davicom DM9161A
AT91RM9200 ETH: 100Mb
... waiting for BOOTP information
Ethernet eth0: MAC address 00:23:31:37:00:1c
IP: 10.0.2.9/255.0.0.0, Gateway: 10.0.0.3
Default server: 10.0.1.2, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [UBOOT]
Non-certified release, version UNKNOWN - built 18:18:48, Jun  8 2006

Platform: DNP/9200 with DNP/EVA9 (ARM9)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.
Copyright (C) 2003, 2004, 2005, 2006 eCosCentric Limited

RAM: 0x20000000-0x22000000, [0x20067040-0x21fdd000] available
FLASH: 0x60000000-0x60ffffff, 128 x 0x20000 blocks
RedBoot>
```

13. Now follow the directions in the previous method to [initialize the flash](#) and [program the ROMRAM RedBoot](#).

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the DNP/9200 with DNP/EVA9 are:

```
$ mkdir redboot_dnp_sk23_romram
$ cd redboot_dnp_sk23_romram
$ ecosconfig new dnp_sk23 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/dnp_sk23/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`.

The other versions of RedBoot - ROM, RAM, JTAG or UBOOT - may be similarly built by choosing the appropriate alternative `.ecm` file.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The DNP/9200 with DNP/EVA9 platform HAL package is loaded automatically when eCos is configured for `dnp_sk23` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The platform HAL package supports three separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x10000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location `0x10000000`. However, when it starts up, the application will first copy itself to RAM at virtual address `0x00000000` and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The DNP/9200 board contains a 16Mbyte Intel 28F128J3 parallel Flash device. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support this part and the platform HAL package contains definitions that customize the driver to the board. This driver is not active until the generic Flash support package, `CYGPKG_IO_FLASH`, is included in the configuration.

Ethernet Driver

The DNP/9200 board uses the AT91RM9200's internal EMAC ethernet device attached to an external Davicom DM9161 PHY. The `CYGPKG_DEVS_ETH_ARM_AT91RM9200` package contains all the code necessary to support this device and the platform HAL package contains definitions that customize the driver to the DNP/9200 board. This driver is not active until the generic Ethernet support package, `CYGPKG_IO_ETH_DRIVERS`, is included in the configuration.

RTC Driver

The DNP/9200 board uses the AT91RM9200's internal RTC support. The `CYGPKG_DEVICES_WALLCLOCK_ARM_AT91` package contains all the code necessary to support this device. This driver is not active until the generic wallclock device support package, `CYGPKG_IO_WALLCLOCK`, is included in the configuration.

Watchdog Driver

The DNP/9200 board uses the AT91RM9200's internal watchdog support. The `CYGPKG_DEVICES_WATCHDOG_ARM_AT91RM9200` package contains all the code necessary to support this device. Within that package the `CYGNUM_DEVS_WATCHDOG_ARM_AT91RM9200_DESIRED_TIMEOUT_MS` configuration option controls the watchdog timeout, and by default will force a reset of the board upon timeout. This driver is not active until the generic watchdog device support package, `CYGPKG_IO_WATCHDOG`, is included in the configuration.

USART Serial Driver

The DNP/9200 board uses the AT91RM9200's internal USART serial support as described in the AT91RM9200 processor HAL documentation. Two serial ports are available: USART 1 which is mapped to virtual vector channel 0 in the HAL diagnostic driver or `"/dev/ser1"` in the interrupt-driven driver; and USART 2 which is mapped to virtual vector channel 1 and `"/dev/ser2"`. Both UARTs support modem control signals such as those used for hardware flow control.

MCI Driver

As described in the SSV board documentation, in order to use the MMC/SD socket on the EVA9 board, the JP8 jumper block must have all jumpers in place, i.e. closed . The SPI jumper block JP9 must have all jumpers removed, i.e. open.

As the AT91RM9200 MCI driver is part of the AT91RM9200 HAL, nothing is required to load it. Similarly the MMC/SD bus driver layer (CYGPKG_DEVS_DISK_MMC) is automatically included as part of the hardware-specific configuration for this target. All that is required to enable the support is to include the generic disk I/O infrastructure package (CYGPKG_IO_DISK), along with the intended filesystem, typically, the FAT filesystem (CYGPKG_FS_FAT) and any of its package dependencies (including CYGPKG_LIBC_STRING and CYGPKG_LINUX_COMPAT for FAT).

Various options can be used to control specific of the AT91RM9200 MCI driver. Consult the AT91RM9200 HAL documentation for information on its configuration.

On this target, it is not possible to detect from the MMC/SD socket whether cards have been inserted or removed. Thus the disk I/O layer's removeable media support will not detect when cards have been inserted or removed, and therefore the only way to detect if a card has been inserted is to attempt mounts.

The MMC/SD socket also does not permit detection of the write-protect (or "lock") switch present on SD cards. "Locked" cards will therefore not be detected which means that despite the switch position, it is still possible to write to them since the lock switch does not physically enforce write protection.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There are just three flags specific to this port:

`-mcpu=arm9`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the AT91RM9200.

`-mthumb`

The arm-elf-gcc compiler will compile C and C++ files into the Thumb instruction set when this option is used. The best way to build eCos in Thumb mode is to enable the configuration option `CYGHWR_THUMB`.

`-mthumb-interwork`

This option allows programs to be created that mix ARM and Thumb instruction sets. Without this option, some memory can be saved. This option should be used if `-mthumb` is used. The best way to build eCos with Thumb interworking is to enable the configuration option `CYGBLD_ARM_ENABLE_THUMB_INTERWORK`.

JTAG debugging support

Name

JTAG support — Usage

Use of JTAG for debugging

JTAG can be used to single-step and debug loaded RAM applications, or even applications resident in ROM, including RedBoot.

Debugging of ROM applications is only possible if using hardware breakpoints. The ARM920T core of the AT91RM9200 only supports two such hardware breakpoints, so they should be used sparingly. If using a GDB front-end such as Eclipse, check it has not set unnecessary extra breakpoints. Some JTAG devices give the option of whether to set hardware or software breakpoints by default. Be sure to configure your device appropriately.

Abatron BDI2000 notes

On the Abatron BDI2000, the `bdi2000.dnp_sk23.cfg` file should be used to setup and configure the hardware to an appropriate state to load programs. This includes setting up the SDRAM controller.

The `bdi2000.dnp_sk23.cfg` file also contains an option to define whether hardware or software breakpoints are used by default, using the `BREAKMODE` directive in the `[TARGET]` section. Edit this file if you wish to use software break points, and remember to use the **boot** command on the BDI2000 command line interface to make the changes take effect.

On the BDI2000, debugging can be performed either via the telnet interface or using **arm-elf-gdb** and the `bdiGDB` interface. In the case of the latter, **arm-elf-gdb** needs to connect to TCP port 2001 on the BDI2000's IP address. For example:

```
(gdb) target remote 111.222.333.444:2001
```

By default when the BDI2000 is powered up, the target will always run the initialization section of the `bdi2000.dnp_sk23.cfg` file (which configures the SDRAM among other things), and halts the target. This behavior is repeated with the **reset halt** command.

If the board is reset when in '**reset halt**' mode (either with the '**reset halt**' or '**reset**' commands, or by pressing the reset button) and the '**go**' command is then given, then the board will boot as normal. If a ROMRAM RedBoot is resident in Flash, it will be run.

It is also possible for the target to always run, without initialization, after the reset button has been pressed. This mode is selected with the **reset run** command. This conveniently allows the target to be connected to the JTAG debugger, and be able to reset it with the reset button, without being required to always type '**go**' every time. Thereafter, invoking the **reset** command will repeat the previous reset style. Also in this mode, exceptions will be handled by board software, rather than causing the JTAG debugger to halt the CPU.

[Suitably configured](#) RAM applications can be loaded either via GDB, or directly via the telnet CLI. For example:

```
Core#0>load 0x20008000 /test.bin bin
Loading /test.bin , please wait ....
```

```
Loading program file passed  
Core#0>go 0x20008000
```

Consult the BDI2000 documentation for information on other formats.

Configuration of RAM applications

If the JTAG device has initialized the SDRAM, such as by using the `bdi2000.dnp_sk23.cfg` configuration on the BDI2000, RAM applications can be loaded directly into SDRAM without requiring a ROM monitor. This loading can be done directly through the JTAG device, or where supported by the JTAG device, through GDB.

In order to configure the application to support this mode, some configuration settings are required. Firstly `CYGSEM_HAL_USE_ROM_MONITOR` must be disabled. This will also cause the `CYGBLD_HAL_ARM9_ATMEL_AT91RM9200_KITS_LOAD_LOW_RAM` configuration option to be enabled allowing the application to be built with a set of memory layout files that will configure the linker script to set the program load address to be within the physical SDRAM space. Secondly the `CYGDBG_HAL_DIAG_TO_DEBUG_CHAN` option should be enabled in order to prevent HAL diagnostic output being encoded into GDB (\$) packets.

Running RAM applications

Once loaded and running via JTAG, HAL diagnostic output will appear by default on the serial debug port. USART 1 can be chosen instead by setting the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` configuration option in the platform HAL to channel 1.

Warning

If resetting the board using the JTAG device, such as by using the BDI2000 **reset** command, the Ethernet PHY fails to interface correctly with the AT91RM9200, and consequently all subsequent ethernet operations are impossible. Only a reset by pressing the reset button, cycling the power or due to a watchdog timeout will cause the PHY to reset correctly.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the DNP/9200 and DNP/EVA9 hardware, and should be read in conjunction with that specification. The DNP/9200 with DNP/EVA9 platform HAL package complements the ARM architectural HAL, the ARM9 variant HAL and the AT91RM9200 processor HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset, the HAL will initialize or reinitialize most of the on-chip peripherals. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address 0x10000000 of the physical memory space. The HAL uses the MMU to locate it at virtual address 0x60000000 after initialization. It remains accessible at address 0x10000000 but accesses to this address range are uncached.

SDRAM

This is located at address 0x20000000 of the physical memory space. The HAL configures the MMU to retain the SDRAM at virtual address 0x20000000, but in order to assign hardware exception vectors at address 0x00000000, the HAL also uses the MMU to create a clone of this memory at virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x30000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. Memory is required for the MMU tables, and must be aligned on a 16Kbyte boundary. These therefore occupy memory from 0x4000 to 0x8000. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00100000, with the bottom 1MB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x00200000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x70000000. The same memory is also accessible uncached and unbuffered at virtual location 0x70100000 for use by devices. At present this memory is entirely reserved for use by the ethernet interface, since there are problems using external SDRAM for ethernet buffers.

On-chip ROM

This is located at address 0x00100000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x71000000. The same memory is also accessible uncached and unbuffered at virtual location 0x71800000.

USB host port

The USB host port (UHP) registers are located at address 0x00300000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x72800000. Memory accessed at this address is uncached and unbuffered. There is no cached variant.

On-chip Peripheral Registers

These are located at address 0xFF000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

eCos uses the SDRAM, parallel NOR flash, ethernet PHY of the DNP/9200 board. eCos does not currently make any use of any other off-chip peripherals present on this board.

Advanced Interrupt Controller

This port has been designed to exploit benefits of the Advanced Interrupt Controller of the AT91RM9200, using the facilities of the AT91RM9200 processor HAL. Consult the documentation in that package for details.

Real-time characterization

The `tm_basic` kernel test gives statistics gathered about the real-time characterization and performance of the kernel. The sample output is shown here for information. The test was built in ARM mode, which provided better performance than Thumb mode.

```
Startup, main stack : stack used    416 size  3920
Startup : Interrupt stack used    524 size  4096
Startup : Idlethread stack used    92 size  2048
```

eCos Kernel Timings

Notes: all times are in microseconds (.000001) unless otherwise stated

```
Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took  13.69 microseconds (6 raw clock ticks)
```

Testing parameters:

```

Clock samples:      32
Threads:            64
Thread switches:    128
Mutexes:            32
Mailboxes:          32
Semaphores:         32
Scheduler operations: 128
Counters:           32
Flags:              32
Alarms:             32

```

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
=====	=====	=====	=====	=====	=====	=====
13.99	8.55	19.23	2.40	43%	26%	Create thread
2.10	0.00	4.27	0.13	95%	3%	Yield thread [all suspended]
2.64	2.14	6.41	0.78	78%	78%	Suspend [suspended] thread
2.90	2.14	4.27	0.98	64%	64%	Resume thread
3.71	2.14	10.68	0.98	67%	31%	Set priority
0.93	0.00	2.14	1.05	56%	56%	Get priority
8.11	6.41	19.23	0.96	70%	28%	Kill [suspended] thread
2.10	0.00	4.27	0.13	95%	3%	Yield [no other] thread
4.17	2.14	6.41	0.25	92%	6%	Resume [suspended low prio] thread
2.94	2.14	6.41	1.03	64%	64%	Resume [runnable low prio] thread
3.60	2.14	6.41	0.96	65%	32%	Suspend [runnable] thread
2.14	0.00	4.27	0.07	96%	1%	Yield [only low prio] thread
2.77	2.14	4.27	0.89	70%	70%	Suspend [runnable->not runnable]
7.68	6.41	19.23	1.23	50%	48%	Kill [runnable] thread
6.51	6.41	12.82	0.19	98%	98%	Destroy [dead] thread
10.75	8.55	21.37	0.46	87%	7%	Destroy [runnable] thread
15.19	12.82	25.64	0.57	90%	3%	Resume [high priority] thread
5.57	4.27	10.68	1.06	58%	40%	Thread switch
0.48	0.00	2.14	0.75	77%	77%	Scheduler lock
1.74	0.00	2.14	0.65	81%	18%	Scheduler unlock [0 threads]
1.78	0.00	2.14	0.59	83%	16%	Scheduler unlock [1 suspended]
1.78	0.00	2.14	0.59	83%	16%	Scheduler unlock [many suspended]
1.74	0.00	2.14	0.65	81%	18%	Scheduler unlock [many low prio]
0.80	0.00	2.14	1.00	62%	62%	Init mutex
2.60	2.14	6.41	0.76	81%	81%	Lock [unlocked] mutex
3.21	2.14	8.55	1.20	96%	56%	Unlock [locked] mutex
2.54	2.14	4.27	0.65	81%	81%	Trylock [unlocked] mutex
2.34	2.14	4.27	0.36	90%	90%	Trylock [locked] mutex
0.47	0.00	2.14	0.73	78%	78%	Destroy mutex
13.09	12.82	19.23	0.50	93%	93%	Unlock/Lock mutex
1.40	0.00	2.14	0.96	65%	34%	Create mbox
0.47	0.00	2.14	0.73	78%	78%	Peek [empty] mbox
3.00	2.14	4.27	1.03	59%	59%	Put [first] mbox
0.67	0.00	2.14	0.92	68%	68%	Peek [1 msg] mbox
3.00	2.14	4.27	1.03	59%	59%	Put [second] mbox
0.80	0.00	2.14	1.00	62%	62%	Peek [2 msgs] mbox
3.07	2.14	4.27	1.05	56%	56%	Get [first] mbox

3.14	2.14	6.41	1.13	56%	56%	Get [second] mbox
2.60	2.14	4.27	0.73	78%	78%	Tryput [first] mbox
2.80	2.14	4.27	0.92	68%	68%	Peek item [non-empty] mbox
3.00	2.14	6.41	1.08	62%	62%	Tryget [non-empty] mbox
2.54	2.14	4.27	0.65	81%	81%	Peek item [empty] mbox
2.47	2.14	4.27	0.56	84%	84%	Tryget [empty] mbox
0.80	0.00	2.14	1.00	62%	62%	Waiting to get mbox
1.00	0.00	2.14	1.06	53%	53%	Waiting to put mbox
3.34	2.14	6.41	1.13	50%	46%	Delete mbox
6.61	6.41	12.82	0.38	96%	96%	Put/Get mbox
0.73	0.00	2.14	0.96	65%	65%	Init semaphore
2.20	0.00	4.27	0.26	90%	3%	Post [0] semaphore
2.40	2.14	4.27	0.47	87%	87%	Wait [1] semaphore
2.20	2.14	4.27	0.13	96%	96%	Trywait [0] semaphore
1.34	0.00	2.14	1.00	62%	37%	Trywait [1] semaphore
0.87	0.00	2.14	1.03	59%	59%	Peek semaphore
0.67	0.00	2.14	0.92	68%	68%	Destroy semaphore
8.75	8.55	14.96	0.38	96%	96%	Post/Wait semaphore
1.20	0.00	2.14	1.05	56%	43%	Create counter
0.40	0.00	2.14	0.65	81%	81%	Get counter value
0.67	0.00	2.14	0.92	68%	68%	Set counter value
2.60	2.14	4.27	0.73	78%	78%	Tick counter
1.00	0.00	2.14	1.06	53%	53%	Delete counter
0.73	0.00	2.14	0.96	65%	65%	Init flag
2.34	2.14	6.41	0.37	93%	93%	Destroy flag
1.87	0.00	2.14	0.47	87%	12%	Mask bits in flag
2.40	2.14	4.27	0.47	87%	87%	Set bits in flag [no waiters]
3.40	2.14	6.41	1.11	53%	43%	Wait for flag [AND]
3.27	2.14	4.27	1.06	53%	46%	Wait for flag [OR]
3.47	2.14	6.41	1.08	56%	40%	Wait for flag [AND/CLR]
3.40	2.14	6.41	1.11	53%	43%	Wait for flag [OR/CLR]
0.47	0.00	2.14	0.73	78%	78%	Peek on flag
1.47	0.00	4.27	1.01	62%	34%	Create alarm
4.41	2.14	8.55	0.38	90%	3%	Initialize alarm
2.27	2.14	4.27	0.25	93%	93%	Disable alarm
4.21	2.14	8.55	0.39	87%	9%	Enable alarm
2.67	2.14	4.27	0.80	75%	75%	Delete alarm
2.74	2.14	4.27	0.86	71%	71%	Tick counter [1 alarm]
14.82	12.82	17.09	0.38	87%	9%	Tick counter [many alarms]
4.87	4.27	6.41	0.86	71%	71%	Tick & fire counter [1 alarm]
86.14	85.47	87.61	0.92	68%	68%	Tick & fire counters [>1 together]
16.89	14.96	19.23	0.48	84%	12%	Tick & fire counters [>1 separately]
10.77	10.68	14.96	0.16	96%	96%	Alarm latency [0 threads]
12.85	12.82	17.09	0.06	99%	99%	Alarm latency [2 threads]
14.06	10.68	17.09	1.21	46%	0%	Alarm latency [many threads]
21.45	21.37	32.05	0.16	99%	99%	Alarm -> thread resume latency
2.20	2.14	6.41	0.00			Clock/interrupt latency
6.18	4.27	10.68	0.00			Clock DSR latency

```
12      0      296 (main stack: 1392) Thread stack used (1360 total)
      All done, main stack : stack used 1392 size 3920
      All done : Interrupt stack used 208 size 4096
      All done : Idlethread stack used 276 size 2048

Timing complete - 30200 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

Other Issues

The DNP/9200 with DNP/EVA9 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The AT91RM9200 processor HAL, ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXIV. Motorola MX1ADS/A Board Support

Overview

Name

eCos Support for the MX1ADS/A Board — Overview

Description

This document covers the Motorola MX1ADS/A single board computer based on the Motorola MC9328MX1. This platform is both hardware and software compatible with the MXLADS/A, which contains a MC9328MXL; all references to the MX1ADS/A and MC9328MX1 should also be taken to refer to the MXLADS/A and MC9328MXL except where explicitly stated.

The MX1ADS/A contains the MC9328MX1 processor, 64Mb of SDRAM, 32MB of flash memory, a CS8900A ethernet MAC, connections for two serial channels and the various other peripherals supported by the MC9328MX1. The MX1ADS/A is identical to the MXLADS/A except that in addition to the devices supported by the MXLADS/A, it contains support for Bluetooth, analogue signal processing and SIM cards. However, since eCos does not use these devices, there is no difference in the support available.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of two Am29PDL127H devices in parallel, giving 256 blocks of 128k bytes each. In a typical setup, the first flash block is used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 254 blocks between 0x10020000 and 0x11FDFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_MC9328MXL` which supports the two UART serial devices on the MC9328MX1. This is configured for the MX1ADS/A by the `CYGPKG_IO_SERIAL_ARM_MX1ADS_A` package. These devices can be used by RedBoot for communication with the host. If either of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver package is loaded automatically when configuring for the MX1ADS/A target.

There is an ethernet driver `CYGPKG_DEVS_ETH_CL_CS8900A` for the Cirrus Logic CS8900A ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_MX1ADS_A` is responsible for configuring this generic driver to the MX1ADS/A hardware. These drivers are also loaded automatically when configuring for the MX1ADS/A target.

eCos manages the on-chip interrupt controller. General Purpose Timer 1 is used to implement the eCos system clock and the microsecond delay function. The Watchdog Timer is also supported. Other on-chip devices (Caches, GPIO, UARTs, memory and interrupt controllers) are initialized only as far as is necessary for eCos to run. Other devices (SPI, I2C, RTC etc.) are not touched.

Tools

The MX1ADS/A port is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.3.3, arm-elf-gdb version 6.1, and binutils version 2.14.

Setup

Name

Setup — Preparing the MX1ADS/A board for eCos Development

Overview

In a typical development environment, the MX1ADS/A board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
SRAM	RedBoot running from RAM, but loaded via the JTAG interface	redboot_SRAM.ecm	redboot_SRAM.bin
RAM	RedBoot running from RAM, usually loaded by another version of RedBoot	redboot_RAM.ecm	redboot_RAM.bin
ROM	RedBoot running from ROM	redboot_ROM.ecm	redboot_ROM.bin
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	redboot_ROMRAM.ecm	redboot_ROMRAM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image.

There are two possible mechanisms for doing this, both via the JTAG interface. The first uses the ability of some JTAG debuggers to write directly to FLASH. The second merely uses the JTAG debugger to load a version of RedBoot and to then use that to program the FLASH.

Direct FLASH Programming

The following instructions describe how to install RedBoot by programming the FLASH directly from the JTAG debugger. At present this has only been tried using an Abatron BDI2000 debugger and these instructions apply to that device. However, it should be possible to adapt these instructions to any other device.

The BDI2000 configuration file needs to be set up to initialize the SDRAM and to tell it what type of FLASH device is present. The resulting config file is shown below:

```
; bdiGDB configuration for Motorola M9328MX1ADS board
; -----
;
[INIT]
;Init SDRAM 16Mx16x2 IAM0 CS2 CL2
;
WM32      0x00221000  0x92120200  ;Set Precharge Command
WM32      0x08200000  0x00000000  ;Issue Precharge all Command
WM32      0x00221000  0xa2120200  ;Set AutoRefresh Command
WM32      0x08000000  0x00000000  ;Issue AutoRefresh Command
WM32      0x08000000  0x00000000
WM32      0x08000000  0x00000000
WM32      0x08000000  0x00000000
WM32      0x08000000  0x00000000
WM32      0x08000000  0x00000000
WM32      0x08000000  0x00000000
WM32      0x08000000  0x00000000
WM32      0x00221000  0xb2120200  ;Set Mode Register
WM32      0x08111800  0x00000000  ;Issue Mode Register Command, Burst Length = 8
WM32      0x00221000  0x82124200  ;Set to Normal Mode
;

[TARGET]
CPUTYPE    ARM920T
CLOCK      1                ;JTAG clock (0=Adaptive, 1=8MHz, 2=4MHz, 3=2MHz)
WAKEUP     3000             ;because of slow rising reset line
RESET      HARD 1000        ;because of heavy capacitive load on reset line
ENDIAN     LITTLE           ;memory model (LITTLE | BIG)
BREAKMODE  HARD
VECTOR     CATCH 0x1f       ;catch D_Abort, P_Abort, SWI, Undef and Reset

[HOST]

[FLASH]
; Program RedBoot with:
; Core#0> erase
; Core#0> prog 0x10000000 MX1 BIN
WORKSPACE  0x08000000
CHIPTYPE   AM29DX32
CHIPSIZE   0x01000000
BUSWIDTH   32
ERASE      0x10000000
ERASE      0x10004000
ERASE      0x10008000
ERASE      0x1000c000
ERASE      0x10010000
ERASE      0x10014000
ERASE      0x10018000
ERASE      0x1001c000
ERASE      0x10020000
```

```
[REGS]
FILE regMX1.def
```

The BDI2000 needs to be rebooted to cause it to reload this configuration file. Once this is done connect to the BDI2000 via its telnet port and issue a reset command:

```
Core#0>reset
- TARGET: processing reset request
- TARGET: BDI asserts TRST and RESET
- TARGET: BDI removes TRST
- TARGET: Bypass check 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x1092001D
- TARGET: All ICEBreaker access checks passed
- TARGET: BDI removes RESET
- TARGET: BDI waits for RESET inactive
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

Now ensure that the FLASH is erased. The following command uses the ERASE entries in the configuration file to erase the first 9 blocks in the FLASH.

```
Core#0>erase
Erasing flash at 0x10000000
Erasing flash at 0x10004000
Erasing flash at 0x10008000
Erasing flash at 0x1000c000
Erasing flash at 0x10010000
Erasing flash at 0x10014000
Erasing flash at 0x10018000
Erasing flash at 0x1001c000
Erasing flash at 0x10020000
Erasing flash passed
Core#0>
```

Copy redboot_ROMRAM.bin to the root directory of the same TFTP server used to fetch the configuration file and execute the following command:

```
Core#0>prog 0x10000000 redboot_ROMRAM.bin bin
Programming redboot_ROMRAM.bin , please wait ....
Programming flash passed
Core#0>
```

If this completes successfully then the FLASH has been programmed. You can start RedBoot by issuing the **go** command, or by detaching the BDI2000 and cycling the power switch of the board. You should see the RedBoot startup screen:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1
```

```
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:51:18, Jul 19 2004
```

```
Platform: Motorola MX1ADS/A (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited
```

```
RAM: 0x00000000-0x02000000, [0x0002f778-0x01fdd000] available
FLASH: 0x10000000 - 0x12000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

Download RedBoot

The following instructions describe how to install RedBoot via the JTAG interface by downloading a version of RedBoot to program the FLASH. This is a two stage process, you must first download a RAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory. The following directions are necessarily somewhat general since the specifics depend on the exact JTAG device available, and the software used to drive it.

Connect the JTAG device to the JTAG connector on the MX1ADS/A board and check that the device is functioning correctly. Using 32 bit memory writes, initialize the static memory controller so that the SDRAM and flash are accessible. The following assignments should be made:

```
*(long *)0x00221000 = 0x92120200; // Set Precharge Command
*(long *)0x08200000 = 0x00000000; // Issue Precharge all Command
*(long *)0x00221000 = 0xa2120200; // Set AutoRefresh Command
*(long *)0x08000000 = 0x00000000; // Issue AutoRefresh Command
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x08000000 = 0x00000000;
*(long *)0x00221000 = 0xb2120200; // Set Mode Register
*(long *)0x08111800 = 0x00000000; // Issue Mode Register Command, Burst Length = 8
*(long *)0x00221000 = 0x82124200; // Set to Normal Mode
```

Now load the SRAM redboot binary image from the file `redboot_SRAM.bin` into SDRAM at `0x08040000`. Exactly how you do this depends on the JTAG driver software. Note that it may be easier to load the ELF or SREC files, if supported, since these contain the correct load addresses.

Connect the serial port of a host machine to UART 1 on the MX1ADS/A board and start a terminal emulator (for example **HyperTerminal** on Windows, **minicom** on Linux) set up to communicate at 38400 baud, 8 bits, one stop bit, no parity. Start RedBoot by executing from location `0x08040000`, which should result in RedBoot starting up and emitting this message on the serial channel:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1
```



```
RedBoot(tm) bootstrap and debug environment [SRAM]
Non-certified release, version UNKNOWN - built 15:50:10, Jul 19 2004
```

```
Platform: Motorola MX1ADS/A (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited
```

```
RAM: 0x08000000-0x0c000000, [0x08065ea0-0x0bfdd000] available
FLASH: 0x10000000 - 0x12000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

Now the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use the terminal emulator's Ymodem support to send the file `redboot_ROMRAM.bin`. This should result in something like the following output:

```
Raw file loaded 0x08066000-0x080b900d, assumed entry at 0x08066000
xyzModem - CRC mode, 2659(SOH)/0(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROMRAM.bin
3848755608 118224 redboot_ROMRAM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 118224
POSIX cksum = 3848755608 118224 (0xe5675998 0x0001cdd0)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)?y
*** Initialize FLASH Image System
... Erase from 0x11fe0000-0x12000000: .
... Program from 0x0bfe0000-0x0c000000 at 0x11fe0000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)?y
... Erase from 0x10000000-0x10020000: .
... Program from 0x08066000-0x08086000 at 0x10000000: .
... Erase from 0x11fe0000-0x12000000: .
... Program from 0x0bfe0000-0x0c000000 at 0x11fe0000: .
RedBoot>
```

The MX1ADS/A board may now be disconnected from the JTAG device and reset by cycling the power. It should then display the startup screen for the ROMRAM version of RedBoot:

```
+... waiting for BOOTP information
Ethernet eth0: MAC address 0e:00:00:ea:18:f0
IP: 10.0.0.207/255.255.255.0, Gateway: 10.0.0.3
Default server: 10.0.0.1, DNS server IP: 10.0.0.1

RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 15:51:18, Jul 19 2004

Platform: Motorola MX1ADS/A (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x0002f778-0x01fdd000] available
FLASH: 0x10000000 - 0x12000000, 256 blocks of 0x00020000 bytes each.
RedBoot>
```

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the MX1ADS/A are:

```
$ mkdir redboot_mxlads_a_romram
$ cd redboot_mxlads_a_romram
$ ecosconfig new mxlads_a redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/mxlads_a/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the SRAM version of RedBoot:

```
$ mkdir redboot_mxlads_a_sram
$ cd redboot_mxlads_a_sram
$ ecosconfig new mxlads_a redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/mxlads_a/VERSION/misc/redboot_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the SRAM RedBoot into the ROM will render the board unbootable.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The MX1ADS/A platform HAL package is loaded automatically when eCos is configured for a `mx1ads_a` or `mx1ads_a` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

The MC9328MX1 SoC is supported by a separate HAL, `CYGPKG_HAL_ARM_ARM9_MC9328MXL`, which supports all the devices on the MC9328MX1/L that eCos uses.

Startup

The MX1ADS/A platform HAL package supports four separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x10000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location `0x10000000`. However, when it starts up the application will first copy itself to RAM at `0x00000000` and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

SRAM

This startup type is used for applications that are downloaded via the JTAG interface. The application is loaded into SRAM at location `0x08040000` and started by executing from that address. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization. However, it is assumed that the machine has been set up from the JTAG interface as described earlier for installing RedBoot.

This configuration is primarily present to provide support for installing RedBoot in the FLASH. It has some limitations with regard to functionality since the MMU is not enabled and no exception vectors are installed at location zero, hence no interrupts can be handled.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is supposed to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for a RAM startup, disabled otherwise. It can be manually disabled for a RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The MX1ADS/A board contains two 16 bit AMD Am29PDL127H flash devices arranged in parallel to form a 32 bit wide interface. The `CYGPKG_DEVS_FLASH_AMD_AM29XXXXX` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_ARM_MX1ADS_A` package contains definitions that customize the driver to the MX1ADS/A board.

Ethernet Driver

The MX1ADS/A board contains a Cirrus Logic CS8900A ethernet MAC. The `CYGPKG_DEVS_ETH_CL_CS8900A` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_MX1ADS_A` package contains definitions that customize the driver to the MX1ADS/A board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=arm9`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM920T CPU in the MC9328MXL.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the MX1ADS/A hardware, and should be read in conjunction with that specification. The MX1ADS/A platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that it uses. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM or ROMRAM startup, the HAL will perform additional initialization, setting up the external RAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

For SRAM startup, minimal initialization is performed, and it is assumed that the JTAG device has initialized the hardware as described earlier.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

SDRAM

This is located at address 0x08000000 of the physical memory space. However the HAL uses the MMU to relocate this to virtual address 0x00000000. The same memory is also accessible uncached and unbuffered at virtual location 0x04000000 for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location 0x00040000, with the bottom 256kB reserved for use by RedBoot.

In the SRAM startup configuration, the SDRAM remains at its physical address of 0x08000000, since the MMU is not enabled.

Flash

This is located at address 0x10000000 of the physical memory space. It is mapped by the HAL using the MMU to the same virtual address with caching and the write buffer enabled.

On-chip Peripheral Registers

These are located at address 0x00200000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping for these at 0xA0000000 in the virtual address space.

All hardware addresses in `mc9328mx1.h` give the physical address. To use these, the macro `CYGARC_VIRTUAL_ADDRESS()` should be applied to these. This will yield the correct address depending on the startup type.

Ethernet MAC

The Cirrus Logic CS8900A is addressed by Chip Select 4, and is located at physical address 0x15000000. It is mapped uncached and unbuffered to the same virtual address when the MMU is enabled.

Other Issues

The MX1ADS/A platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXV. ARM Versatile 926EJ-S Board Support

Overview

Name

eCos Support for the Versatile 926EJ-S Board — Overview

Description

This document covers the ARM Versatile Platform Baseboard for the ARM926EJ-S development chip, hereafter referred to as the VPB926EJS. The VPB926EJS contains the ARM926EJ-S processor, 128Mb of SDRAM, 64MB of Intel Strataflash memory, 2Mb of static RAM, a SMSC LAN91C111 Ethernet MAC, and external connections for the three on-chip and one off-chip serial channels, ethernet and the various other peripherals supported by the ARM926EJ-S.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 256 blocks of 256k bytes each. In a typical setup, the first flash block is used for the ROMRAM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 254 blocks between 0x34040000 and 0x37FBFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_ARM_PL011` which supports the ARM PL011 PrimeCell UARTs used by the VPB926EJS. The `CYGPKG_IO_SERIAL_ARM_VPB926EJS` package provides customization of this generic driver to the VPB926EJS hardware. These devices can be used by RedBoot for communication with the host. If any of these devices is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver packages are loaded automatically when configuring for the VPB926EJS target.

There is an ethernet driver `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` for the SMSC LAN91C111 ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_VPB926EJS` is responsible for configuring this generic driver to the VPB926EJS hardware. These drivers are also loaded automatically when configuring for the VPB926EJS target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, UARTs, MPMC, SSMC, I2C etc.) are initialized only as far as is necessary for eCos to run. Other devices (PCI, SSP, SCI, GPIO etc.) are not touched.

Tools

The VPB926EJS port is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.2.1, arm-elf-gdb version 5.3, and binutils version 2.13.1.

Setup

Name

Setup — Preparing the VPB926EJS board for eCos Development

Overview

In a typical development environment, the VPB926EJS board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
RAM	RedBoot running from RAM	redboot_RAM.ecm	redboot_RAM.bin
ROM	RedBoot running from flash ROM	redboot_ROM.ecm	redboot_ROM.bin
ROMRAM	RedBoot running from RAM, but contained in the board's flash boot sector	redboot_ROMRAM.ecm	redboot_ROMRAM.bin
SRAM	RedBoot running from static RAM	redboot_SRAM.ecm	redboot_SRAM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Initial Installation

Flash Installation

Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image. This is a two stage process, you must first download an SRAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory.

The VPB926EJS boards are shipped from ARM with a version of ARM's boot monitor installed. Unfortunately this boot monitor only works in conjunction with ARM's tools, which are not supplied with the board. Hence the only viable approach is to install RedBoot via the JTAG interface. The following directions are necessarily somewhat general since the specifics depend on the exact JTAG device available, and the software used to drive it.

Connect the JTAG device to the JTAG connector on the Versatile board and check that the device is functioning correctly. Using 32 bit memory writes, initialize the static memory controller so that the SRAM and flash are accessible. The following assignments should be made:

```

*(long *)0x10100034 = 0x00303021;
*(long *)0x10100054 = 0x00303021;
*(long *)0x10100074 = 0x00303021;
*(long *)0x10100094 = 0x00303021;

```

Now load the SRAM redboot binary image from the file `redboot_SRAM.bin` into the base of SRAM at `0x38000000`. Exactly how you do this depends on the JTAG driver software.

Start RedBoot by executing from location `0x38000040`, which should result in RedBoot starting up.

```

+... waiting for BOOTP information
Ethernet eth0: MAC address 00:02:f7:00:0b:34
IP: 10.0.0.208/255.255.255.0, Gateway: 10.0.0.1
Default server: 10.0.0.201

RedBoot(tm) bootstrap and debug environment [SRAM]
Non-certified release, version UNKNOWN - built 16:03:09, May  5 2004

Platform: ARM Versatile VPB926EJS (ARM9)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x02000000, [0x00013b20-0x01fbd000] available
FLASH: 0x34000000 - 0x38000000, 256 blocks of 0x00040000 bytes each.
RedBoot>

```

Now the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use HyperTerminal's Ymodem support to send the file `redboot_ROMRAM.bin`. This should result in something like the following output:

```

Raw file loaded 0x0002f800-0x0004be6b, assumed entry at 0x0002f800
xyzModem - CRC mode, 911(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>

```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```

$ cksum redboot_ROMRAM.bin
140216855 116332 redboot_ROMRAM.bin

```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```

RedBoot> cksum -b %{FREEMEMLO} -l 116332
POSIX cksum = 140216855 116332 (0x085b8a17 0x0001c66c)

```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```

RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)?y
*** Initialize FLASH Image System
... Unlock from 0x37fc0000-0x38000000: .
... Erase from 0x37fc0000-0x38000000: .
... Program from 0x03fc0000-0x04000000 at 0x37fc0000: .
... Lock from 0x37fc0000-0x38000000: .
RedBoot> fis create -b %{FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)?y
... Unlock from 0x34000000-0x34040000: .
... Erase from 0x34000000-0x34040000: .
... Program from 0x0002f800-0x0006f800 at 0x34000000: .
... Lock from 0x34000000-0x34040000: .
... Unlock from 0x37fc0000-0x38000000: .
... Erase from 0x37fc0000-0x38000000: .
... Program from 0x03fc0000-0x04000000 at 0x37fc0000: .
... Lock from 0x37fc0000-0x38000000: .
RedBoot>

```

The VPB926EJS board may now be reset either by cycling the power, pressing the reset switch, or with the **reset** command. It should then display the startup screen for the ROMRAM version of RedBoot.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROMRAM version of RedBoot for the VPB926EJS are:

```

$ mkdir redboot_vpb926ejs_romram
$ cd redboot_vpb926ejs_romram
$ ecosconfig new vpb926ejs redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/vpb926ejs/VERSION/misc/redboot_ROMRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make

```

To rebuild the SRAM version of RedBoot:

```

$ mkdir redboot_vpb926ejs_sram
$ cd redboot_vpb926ejs_sram
$ ecosconfig new vpb926ejs redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/vpb926ejs/VERSION/misc/redboot_SRAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make

```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the SRAM RedBoot into the ROM will render the board unbootable.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The VPB926EJS platform HAL package is loaded automatically when eCos is configured for a `vpb926ejs` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The VPB926EJS platform HAL package supports four separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0x34000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

ROMRAM

This startup type can be used for finished applications which will be programmed into flash at physical location `0x34000000`. However, when it starts up, the application will first copy itself to RAM at `0x00000000` and then run from there. RAM is generally faster than flash memory, so the program will run more quickly than a ROM-startup application. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

SRAM

This startup type exists only to support the installation of RedBoot via the JTAG interface. Functionally, it is equivalent to the ROM startup type except that the executable image is loaded into the static RAM at `0x38000000` rather than the flash ROM. It is of little use for other applications.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is required to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for RAM startup, disabled otherwise. It can be manually disabled for RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The VPB926EJS board contains two 32Mb 28F256K3 Intel StrataFlash flash devices, giving 64MB of flash in total. The `CYGPKG_DEVS_FLASH_STRATA_V2` package contains all the code necessary to support these parts and the VPB926EJS platform HAL package contains definitions that customize the driver to the VPB926EJS board.

Ethernet Driver

The VPB926EJS board contains a SMSC LAN91C111 ethernet controller. The `CYGPKG_DEVS_ETH_SMSC_LAN91CXX` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_VPB926EJS` package contains definitions that customize the driver to the VPB926EJS board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

```
-mcpu=arm9
```

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=arm9` is the correct option for the ARM926E CPU.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the VPB926EJS hardware, and should be read in conjunction with that specification. The VPB926EJS platform HAL package complements the ARM architectural HAL and the ARM9 variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that are used by eCos. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM, ROMRAM or SRAM startup, the HAL will perform additional initialization, setting up the external SDRAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Linker Scripts and Memory Maps

The platform HAL package provides the memory layout information needed to generate the linker script. The key memory locations are as follows:

Flash

This is located at address `0x34000000` of the physical memory space. The HAL uses the MMU to map it cached and buffered at the same address.

SDRAM

There are two blocks of SDRAM in the system. One block of 64Mb is present at physical address `0x00000000` and is echoed at `0x04000000`. The second block is present at physical address `0x08000000`. The HAL uses the MMU to map the first block to virtual address `0x00000000` and the second to virtual address `0x04000000`, forming a single contiguous 128Mb block of SDRAM starting at `0x00000000`. The same memory is also accessible uncached and unbuffered at virtual location `0x70000000` for use by devices. The first 32 bytes are used for hardware exception vectors. The next 32 bytes are used for the VSR table and the next 256 bytes are normally used for the eCos virtual vectors, allowing RAM-based applications to use services provided by the ROM monitor. For ROM/ROMRAM startup, all remaining SDRAM is available. For RAM startup, available RAM starts at virtual location `0x00040000`, with the bottom 256kB reserved for use by RedBoot.

On-chip SRAM

This is located at address 0x38000000 of the physical memory space. The HAL uses the MMU to map this to the same virtual address, but cached and buffered. The same memory is also accessible uncached and unbuffered at virtual location 0x78000000 for use by devices. This memory is not used by eCos for any purpose except in the SRAM startup mode, when it contains the system image.

On-chip Peripheral Registers

These are located at address 0x10000000 in the physical memory space. When the MMU is enabled, it sets up a direct, uncached, unbuffered mapping so that these registers remain accessible at their physical locations.

Off-chip Peripherals

All off-chip peripherals are also visible in the 0x10000000 address space.

Other Issues

The VPB926EJS platform HAL does not affect the implementation of other parts of the eCos HAL specification. The ARM9 variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXVI. Intel XScale IXP4xx Network Processor Support

Overview

Name

Support for Intel XScale IXP4xx Network Processors — Overview

Description

This document covers the configuration and usage of the Hardware Abstraction Layer (HAL) for the Intel XScale IXP4xx Network Processor series, including the IXP425 and IXP465. It is expected to be read in conjunction with platform HAL-specific documentation, as well as the eCos HAL specification. This processor HAL package complements the ARM architectural HAL, XScale variant HAL and the platform HAL. It provides functionality common to IXP4xx-based board implementations.

This support is found in the eCos package located at `packages/hal/arm/xscale/ixp425` within the eCos source repository.

For historical reasons many of the definitions, filenames and configuration options in this package refer to the IXP425 specifically. In fact, unless otherwise noted, these definitions, filenames and configuration options apply equally to all members of the IXP4xx family.

The IXP4xx processor HAL package is loaded automatically when eCos is configured for an IXP4xx-based platform. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Supported Hardware

Supported features of the Intel XScale IXP4xx processors within this processor HAL package include:

- [IXP4xx-specific hardware definitions](#)
- [Interrupt controller](#)
- [General-purpose timers](#)
- [Watchdog timer](#)
- [Serial UARTs](#)
- [PCI bus controller](#)
- [PCI bus IDE controllers](#)
- [CompactFlash cards in TrueIDE mode](#)
- [General Purpose I/O \(GPIO\)](#)

For licensing-related reasons, support for the Network Processing Engines (NPEs) at this time is only available with an add-on EPK package from Intel. eCosCentric is unable to provide support for this add-on package.

IXP4xx hardware definitions

Name

IXP4xx hardware definitions — Details on obtaining hardware definitions for IXP4xx

Register definitions

The file `<cyg/hal/hal_ixp425.h>` can be included from application and eCos package sources to provide definitions related to IXP4xx subsystems. These include register definitions for the PCI bus controller, SDRAM controller, DDR controller for IXP46x, expansion bus controller, I²C[®] controller for IXP46x, General purpose I/O (GPIO), interrupt controller, general-purpose timers and watchdog timer.

I/O Definitions

The file `<cyg/hal/var_io.h>` contains definitions used by the PCI support, as well as memory mapping details and macros to read and write the I/O space (including the PCI I/O space) at a variety of widths. If PCI IDE support has been enabled, it also provides the relevant support macros.

IXP4xx interrupt controller

Name

IXP4xx interrupt controller — Interrupt controller definitions and usage

Interrupt controller definitions

The file <cyg/hal/hal_var_ints.h> (located at hal/arm/xscale/ixp425/VERSION/include/hal_var_ints.h in the eCos source repository) contains interrupt vector number definitions for use with the eCos kernel and driver interrupt APIs:

```
#define CYGNUM_HAL_INTERRUPT_NONE          -1
#define CYGNUM_HAL_INTERRUPT_NPEA          0
#define CYGNUM_HAL_INTERRUPT_NPEB          1
#define CYGNUM_HAL_INTERRUPT_NPEC          2
#define CYGNUM_HAL_INTERRUPT_QM1           3
#define CYGNUM_HAL_INTERRUPT_QM2           4
#define CYGNUM_HAL_INTERRUPT_TIMER0         5
#define CYGNUM_HAL_INTERRUPT_GPIO0         6
#define CYGNUM_HAL_INTERRUPT_GPIO1         7
#define CYGNUM_HAL_INTERRUPT_PCI_INT        8
#define CYGNUM_HAL_INTERRUPT_PCI_DMA1       9
#define CYGNUM_HAL_INTERRUPT_PCI_DMA2      10
#define CYGNUM_HAL_INTERRUPT_TIMER1        11
#define CYGNUM_HAL_INTERRUPT_USB           12
#define CYGNUM_HAL_INTERRUPT_UART2         13
#define CYGNUM_HAL_INTERRUPT_TIMESTAMP      14
#define CYGNUM_HAL_INTERRUPT_UART1         15
#define CYGNUM_HAL_INTERRUPT_WDOG           16
#define CYGNUM_HAL_INTERRUPT_AHB_PMU        17
#define CYGNUM_HAL_INTERRUPT_XSCALE_PMU     18
#define CYGNUM_HAL_INTERRUPT_GPIO2         19
#define CYGNUM_HAL_INTERRUPT_GPIO3         20
#define CYGNUM_HAL_INTERRUPT_GPIO4         21
#define CYGNUM_HAL_INTERRUPT_GPIO5         22
#define CYGNUM_HAL_INTERRUPT_GPIO6         23
#define CYGNUM_HAL_INTERRUPT_GPIO7         24
#define CYGNUM_HAL_INTERRUPT_GPIO8         25
#define CYGNUM_HAL_INTERRUPT_GPIO9         26
#define CYGNUM_HAL_INTERRUPT_GPIO10        27
#define CYGNUM_HAL_INTERRUPT_GPIO11        28
#define CYGNUM_HAL_INTERRUPT_GPIO12        29
#define CYGNUM_HAL_INTERRUPT_SW_INT1       30
#define CYGNUM_HAL_INTERRUPT_SW_INT2       31

#ifdef CYGHWL_HAL_ARM_XSCALE_CPU_IXP46x
#define CYGNUM_HAL_INTERRUPT_USB_HOST      32
#define CYGNUM_HAL_INTERRUPT_I2C           33
#define CYGNUM_HAL_INTERRUPT_SPI           34
#define CYGNUM_HAL_INTERRUPT_TIMESYNC      35
#define CYGNUM_HAL_INTERRUPT_EAU_DONE      36
```

```
#define CYGNUM_HAL_INTERRUPT_SHA_DONE      37
#define CYGNUM_HAL_INTERRUPT_SWCP_PERR     58
#define CYGNUM_HAL_INTERRUPT_QMGR_PERR     60
#define CYGNUM_HAL_INTERRUPT_MCU_ERR       61
#define CYGNUM_HAL_INTERRUPT_EXP_PERR      62
#endif
```

The list of interrupt vectors may be augmented on a per-platform basis. Consult the platform HAL documentation for your platform for whether this is the case.

Interrupt controller functions

The source file `src/ixp425_misc.c` within this package provides most of the support functions to manipulate the interrupt controller. The `hal_irq_handler` queries the IRQ status register to determine the interrupt cause. Functions `hal_interrupt_mask` and `hal_interrupt_unmask` enable or disable interrupts within the interrupt controller.

GPIO interrupts are configured in the `hal_interrupt_configure` function, where the `level` and `up` arguments are interpreted as follows:

level	up	interrupt on
0	0	Falling Edge
0	1	Rising Edge
0	-1	Either Edge
1	0	Low Level
1	1	High Level

Some interrupts are acknowledged in the `hal_interrupt_acknowledge` function, although for many of the IXP4xx on-chip peripherals, the means to stop the interrupt from triggering again is to remove the cause of the interrupt in a device dependent way. This function does however clear GPIO interrupts.

Macros of the following forms:

```
HAL_PLF_INTERRUPT_MASK(vector)
HAL_PLF_INTERRUPT_UNMASK(vector)
HAL_PLF_INTERRUPT_ACKNOWLEDGE(vector)
HAL_PLF_INTERRUPT_CONFIGURE(vector)
```

may be defined by the platform HAL via the header file defined by its `CYGBLD_HAL_PLATFORM_H` macro definition in order to extend support to further vectors.

Note that in all the above, it is not recommended to call the described functions directly. Instead either the HAL macros (`HAL_INTERRUPT_MASK` et al) or preferably the kernel or driver APIs should be used to control interrupts.

Interrupt handling withing standalone applications

For non-eCos standalone applications running under RedBoot, it is possible to install an interrupt handler into the interrupt vector table manually. Memory mappings are platform-dependent and so the platform documentation should be consulted, but in general the address of the interrupt table can be determined by analyzing RedBoot's symbol table, and searching for the address of the symbol name `hal_interrupt_handlers`. Table slots correspond to the interrupt numbers [above](#). Pointers inserted in this table should be pointers to a C/C++ function with the following prototype:

```
extern unsigned int isr( unsigned int vector, unsigned int data );
```

For non-eCos applications run from RedBoot, the return value can be ignored. The `vector` argument will also be the [interrupt vector number](#). The `data` argument is extracted from a corresponding table named `hal_interrupt_data` which immediately follows the interrupt vector table. It is still the responsibility of the application to enable and configure the interrupt source appropriately if needed.

IXP46x

Support exists for the IXP46x in order to query and manipulate the extended interrupt controller registers handling interrupt numbers of 32 and above. Also on the IXP46x an interrupt handler is attached to handle ECC errors from the MCU.

General-purpose timers

Name

General-purpose timers — Use of IXP4xx general-purpose timers

General-purpose timer 0

The IXP4xx processor HAL provides general-purpose timer 0 (OST_TIM0) to the eCos kernel for use as a real-time clock. This timer is also used for implementing the HAL_DELAY_US functionality, which is used by some device drivers, and in non-kernel configurations such as with RedBoot where this timer is needed for loading program images via X/Y-modem protocols and debugging via TCP/IP. Standalone applications which require RedBoot services, such as debugging, should avoid use of this timer.

Available timers

General-purpose timer 1 and the timestamp timer are available for application use.

System clock configuration

By default for eCos applications, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option CYGNUM_HAL_RTC_DENOMINATOR which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed. If the desired frequency cannot be expressed accurately solely with changes to CYGNUM_HAL_RTC_DENOMINATOR, then the configuration option CYGNUM_HAL_RTC_NUMERATOR may also be adjusted, and again clock-related settings will automatically be recalculated.

Watchdog

Name

Watchdog — Describes use of the hardware watchdog

Description

This HAL package includes the hardware driver required to support the generic watchdog support defined in the `CYGPKG_IO_WATCHDOG` package. The functionality allows the watchdog either to cause the IXP4xx to reset, or to cause a callback function to be triggered as defined in the generic watchdog API.

The watchdog is also used to perform platform resets, such as with the **reset** command in the RedBoot CLI. Note that there is an IXP425 erratum affecting use of the watchdog on IXP425s with stepping level of A0. Stepping levels of B0 and above no longer have this issue.

Configuration

It is not possible to enable support for the IXP4xx on-chip watchdog unless the `CYGPKG_IO_WATCHDOG` package is included in the eCos configuration. Once included, the configuration options for this hardware support can then be found in the watchdog part of the eCos configuration tree under the CDL component name `CYGPKG_HAL_IXP4XX_WATCHDOG`, rather than in the HAL.

The option `CYGNUM_HAL_IXP4XX_WATCHDOG_DESIRED_TIMEOUT_US` allows the watchdog timeout interval to be set. An application must ensure that the watchdog is reset more frequently than this period. The units are microseconds, and depending on the selected period, values may be rounded up to the next clock tick..

If the option `CYGSEM_HAL_IXP4XX_WATCHDOG_RESET` is enabled, watchdog expiration will cause the IXP4xx to perform a soft reset. If this option is instead disabled, a callback function can be registered with the generic watchdog API, which will be called by an interrupt handler associated with the watchdog.

Serial UARTs

Name

Serial UARTs — Configuration and implementation details of serial UART support

Overview

There are two forms of support for the built-in high-speed and console serial UARTs. In all cases the default serial port settings are 115200,8,N,1 with no flow control.

HAL diagnostic I/O

One form is polled mode HAL diagnostic output, intended primarily for use during debug and development. Operations are usually performed with global interrupts disabled, and thus is not usually suitable for deployed systems. This can operate on either port, according to the configuration settings.

Several configuration options within the IXP4xx processor HAL affect HAL diagnostic operation. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL` selects the serial port channel to use as the console at startup time. Channel 0 is the debug serial port, channel 1 is the high-speed serial port. This will be the channel that receives output from, for example, `diag_printf()`. The CDL option `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL` selects the serial port channel to use for GDB communication by default. Note that when using RedBoot, these options are usually inactive as it is RedBoot that decides which channels are used. Applications may override RedBoot's selections by enabling the `CYGSEM_HAL_VIRTUAL_VECTOR_CLAIM_COMMS` CDL option in the HAL. Baud rates for each channel are set with the `CYGNUM_HAL_VIRTUAL_VECTOR_CONSOLE_CHANNEL_BAUD` and `CYGNUM_HAL_VIRTUAL_VECTOR_DEBUG_CHANNEL_BAUD` options.

Interrupt-driven serial driver

The second form of support is an interrupt-driven serial driver, which is integrated into the eCos standard serial I/O infrastructure (`CYGPKG_IO_SERIAL`). This support can be enabled on either port. Note that if this form of serial driver is enabled on a port, it will prevent ctrl-c operation when debugging.

This driver uses the eCos generic 16x5x UART driver found in `CYGPKG_IO_SERIAL_GENERIC_16X5X` to provide most of the support. That driver package should also be consulted for documentation and configuration options.

Note that a standard 16550A compatible UART has receive FIFO trigger levels that can be set with the `CYGPKG_IO_SERIAL_GENERIC_16X5X_FIFO_RX_THRESHOLD` to 1, 4, 7, 8 or 14 bytes. However the IXP4xx family has a larger 64-byte FIFO, and so these values should be mapped to 1, 8, 16 or 32. In other words setting the `CYGPKG_IO_SERIAL_GENERIC_16X5X_FIFO_RX_THRESHOLD` option to 8 bytes would in fact cause the receive FIFO to trigger at 16 bytes and so forth.

The maximum baud rate supported by the generic eCos infrastructure is 230,400 and so at this time the higher baud rates of 460,800 and 921,600 baud are not supported.

PCI bus controller

Name

PCI bus controller — PCI bus controller support implementation details

Implementation details

The PCI bus controller is supported with the files `<cyg/hal/var_io.h>` and the source file `src/ixp425_pci.c`. These files provide almost all the required underlying support for use with the generic PCI package `CYGPKG_IO_PCI`.

Platform HALs are still required to provide any extra initialisation steps such as pin routing, involving GPIO, particularly for the interrupt pins. Accordingly any such interrupt pins will also require treatment with an implementation of the following function:

```
extern void cyg_hal_plf_pci_translate_interrupt(cyg_uint32 bus, cyg_uint32 devfn,  
                                              CYG_ADDRWORD *vec, cyg_bool *valid);
```


PCI bus IDE controllers

Name

PCI bus IDE controllers — Configuring and using IDE controllers on the PCI bus

Overview

Support is included for IDE bus controllers connected via the PCI bus. This includes support for generic PCI IDE controllers as well as Promise 20275 or 20269 IDE controllers.

Configuration

This support can be enabled with the `CYGFUN_HAL_IXP4XX_PCI_IDE_SUPPORT` configuration option and is mutually exclusive with support for TrueIDE mode disks connected via a CompactFlash interface.

Use with RedBoot

Enabling this option allows RedBoot to be used to load program images off an IDE disk connected via a PCI IDE controller, for example a Linux kernel from an EXT2 partition of an IDE disk using the "disk" load method.

Files

The file `src/ixp_pci_ide.c` within this package is used to provide the necessary underlying support.

CompactFlash cards in TrueIDE mode

Name

CompactFlash cards in TrueIDE mode — Using CompactFlash cards in TrueIDE mode on the IXP4xx expansion bus

Overview

The IXP4xx processor HAL includes support for CompactFlash IDE devices accessed in True IDE mode directly on the IXP4xx expansion bus.

The hardware configuration for attaching the CF IDE devices must follow the specification described in Intel Application Note 30245603: *Intel IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor: Using CompactFlash* (<http://www.intel.com/design/network/applnots/302456.htm>). This describes use of the EXP_CS_N_1 (CS1) and EXP_CS_N_2 (CS2) signals to control the CF IDE device.

Configuration

Support for CF True IDE devices is contained within the IXP4xx variant HAL and is controlled with the CYGFUN_HAL_IXP4XX_CF_TRUE_IDE_SUPPORT configuration option. It is not possible to include PCI IDE support within the same configuration - eCos only allows one ID controller driver at the present time.

The driver requires the presence of the generic disk layer (CYGPKG_IO_DISK), as well as the IDE disk driver (CYGPKG_DEVS_DISK_IDE).

The block device name used to identify the disk is configured with the option CYGDAT_HAL_IXP4XX_CF_TRUE_IDE_DISK_NAME and defaults to /dev/hd0/. An MBR is expected to be present on the CF card, and individual partitions can be accessed as e.g. /dev/hd0/1, /dev/hd0/2, etc. or the whole device as /dev/hd0/0.

With this support it is possible to access filesystems on the CF IDE card with the further inclusion of the generic file I/O layer package (CYGPKG_IO_FILEIO) along with a standard eCos filesystem implementation such as FAT (CYGPKG_FS_FAT).

Use from RedBoot

Similarly, it is possible for RedBoot to load images from a filesystem using the "file" load method. For example:

```
RedBoot> fs mount -d /dev/hd0/1 -t fatfs
RedBoot> fs list
   2 -rw-rw-rw-rw  1 size 3961588 VMLINUX
 1937 -rw-rw-rw-rw  1 size 1588984 ZIMAGE
RedBoot> load -m file -r -b %{freememlo} /ZIMAGE
Raw file loaded 0x00079800-0x001fd6f7, assumed entry at 0x00079800
RedBoot> exec
Using base address 0x00079800 and length 0x00183ef8
Uncompressing Linux.....
```

CompactFlash cards in TrueIDE mode

```
Linux version 2.6.12 (root@andy) (gcc version 3.4.4) #43 Tue Nov 15 16:40:04 GMT 2005
CPU: XScale-IXP42x Family [690541c1] revision 1 (ARMv5TE)
CPU0: D VIVT undefined 5 cache
CPU0: I cache: 32768 bytes, associativity 32, 32 byte lines, 32 sets
CPU0: D cache: 32768 bytes, associativity 32, 32 byte lines, 32 sets
Machine: Intel IXDP425 Development Platform
[etc.]
```

Implementation details

The implementation assumes that the platform HAL will map memory accesses at 0x51000000 to expansion bus accesses with CS1 enabled, and similarly 0x52000000 to expansion bus accesses with CS2 enabled.

This driver operates in polled mode (PIO) only with no interrupt-driven operation nor DMA, and uses conservative bus configuration timings to allow for maximum compatibility with CF IDE cards. Note that some CF cards are not fully compliant with the CompactFlash standard and do not fully or correctly implement True IDE mode.

GPIO

Name

GPIO — General purpose I/O

GPIO functions

As well as hardware definitions, the file `<cyg/hal/hal_ixp425.h>` provides a set of macros to assist GPIO manipulation:

```
HAL_GPIO_OUTPUT_ENABLE(line)
HAL_GPIO_OUTPUT_DISABLE(line)
HAL_GPIO_OUTPUT_SET(line)
HAL_GPIO_OUTPUT_CLEAR(line)
```

As described [earlier](#), HAL support already exists for handling GPIO lines configured as interrupts.

XXXVII. Intel IQ80321 Board Support

Overview

Name

eCos Support for the Intel IQ80321 Board — Overview

Description

This document covers the Intel IQ80321 development board for the IOP321 XScale device. The IQ80321 contains the IOP321 processor, 128MB of SDRAM, 8MB of Intel Strataflash memory, an Intel i82544 Ethernet MAC and a TL16C550C UART.

For typical eCos development, a RedBoot image is programmed into the flash memory, and the board will boot this image from reset. RedBoot provides gdb stub functionality so it is then possible to download and debug stand-alone and eCos applications via the gdb debugger. This can happen over either a serial line or over ethernet.

Supported Hardware

The flash memory consists of 64 blocks of 128k bytes each. In a typical setup, the first two flash blocks are used for the ROM RedBoot image. The topmost block is used to manage the flash and hold RedBoot **fconfig** values. The remaining 62 blocks between 0xF0040000 and 0xF07DFFFF can be used by application code.

There is a serial driver `CYGPKG_IO_SERIAL_GENERIC_16X5X` which supports the 16C550 UART. The `CYGPKG_IO_SERIAL_ARM_IQ80321` package provides customization of this generic driver to the IQ80321 hardware. This device can be used by RedBoot for communication with the host. If this device is needed by the application, either directly or via the serial driver, then it cannot also be used for RedBoot communication. Another communication channel such as ethernet should be used instead. The serial driver packages are loaded automatically when configuring for the IQ80321 target.

There is an ethernet driver `CYGPKG_DEVS_ETH_INTEL_I82544` for the Intel i82544 ethernet device. A second package `CYGPKG_DEVS_ETH_ARM_IQ80321` is responsible for configuring this generic driver to the IQ80321 hardware. These drivers are also loaded automatically when configuring for the IQ80321 target.

eCos manages the on-chip interrupt controller. Timer 0 is used to implement the eCos system clock and the microsecond delay function. Other on-chip devices (Caches, MEMC, INTC, PCI bridge, ATU, MMU, I2C) are initialized only as far as is necessary for eCos to run. Other devices are not touched.

Tools

The IQ80321 port is intended to work with GNU tools configured for an arm-elf target. The original port was undertaken using arm-elf-gcc version 3.2.1, arm-elf-gdb version 5.3, and binutils version 2.13.1.

Setup

Name

Setup — Preparing the IQ80321 board for eCos Development

Overview

In a typical development environment, the IQ80321 board boots from flash into the RedBoot ROM monitor. eCos applications are configured for RAM startup and then downloaded and run on the board via the debugger **arm-elf-gdb**. Preparing the board therefore usually involves programming a suitable RedBoot image into flash memory.

The following RedBoot configurations are supported:

Configuration	Description	Use	File
ROM	RedBoot running from flash ROM boot sector	redboot_ROM.ecm	redboot_ROM.bin
RAM	RedBoot running from RAM with RedBoot in the flash boot sector	redboot_RAM.ecm	redboot_RAM.bin

For serial communications, all versions run with 8 bits, no parity, and 1 stop bit at 38400 baud. RedBoot also supports ethernet communication and flash management.

Hardware Setup

The 80321 board is highly configurable through a number of switches and jumpers. RedBoot and eCos make some assumptions about board configuration and attention must be paid to these assumptions for reliable operation:

- The onboard ethernet and the secondary slot may be placed in a private space so that they are not seen by a PC BIOS. If the board is to be used in a PC with BIOS, then the ethernet should be placed in this private space so that RedBoot/eCos and the BIOS do not conflict.
- RedBoot assumes that the board is plugged into a PC with BIOS. This requires RedBoot to detect when the BIOS has configured the PCI-X secondary bus. If the board is placed in a backplane, RedBoot will never see the BIOS configure the secondary bus. To prevent this wait, set switch S7E1-3 to ON when using the board in a backplane.
- For the remaining switch settings, the following is a known good configuration:

S1D1	All OFF
S7E1	7 is ON, all others OFF
S8E1	2,3,5,6 are ON, all others OFF
S8E2	2,3 are ON, all others OFF
S9E1	3 is ON, all others OFF

S4D1	1,3 are ON, all others OFF
J9E1	2,3 jumpered
J9F1	2,3 jumpered
J3F1	Nothing jumpered
J3G1	2,3 jumpered
J1G2	2,3 jumpered

Initial Installation

Flash Installation

The IQ80321 is supplied with a version of RedBoot in flash. It is recommended that this version of RedBoot be replaced with one that is built from the same set of sources as the eCos system to be run on it. There are several ways of doing this.

The board manufacturer provides a DOS application which is capable of programming the flash over the PCI bus, and this is required for initial installations of RedBoot. Please see the board manual for information on using this utility. In general, the process involves programming the ROM mode RedBoot image to flash. RedBoot should be programmed to flash address 0x00000000 using the DOS utility.

If a JTAG debugger is available (such as the Abatron BDI2000) that supports programming the flash, then this may be used to install the new RedBoot. See the documentation for the JTAG device to find out how to initialize the board and program the flash. RedBoot needs to be programmed to flash address 0x00000000.

Finally, RedBoot may be installed by using the resident RedBoot. Installing RedBoot is a matter of downloading a new binary image and overwriting the existing Boot monitor ROM image. This is a two stage process, you must first download a RAM-resident version of RedBoot and then use that to download the ROM image to be programmed into the flash memory.

Connect to RedBoot as described in the IQ80321 documentation using a terminal emulator (for example **HyperTerminal** on Windows, **minicom** on Linux). You should see the RedBoot startup banner, similar to the following:

```
RedBoot(tm) bootstrap and debug environment [ROM]

Platform: IQ80321 (XScale)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.

RAM: 0x00000000-0x08000000, [0x0001b068-0x07fd1000] available
FLASH: 0xf0000000 - 0xf0800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

The RAM image can be downloaded using the following RedBoot command:

```
RedBoot> load -m ymodem
```

Use the terminal emulator's Ymodem support to send the file `redboot_RAM.srec`. This should result in something like the following output:

```
Entry point: 0x00020040, address range: 0x00020000-0x000479f8
xyzModem - CRC mode, 2(SOH)/456(STX)/0(CAN) packets, 5 retries
RedBoot>
```

Now start the RAM version of RedBoot:

```
RedBoot> go
+No network interfaces found

RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version v2_0_24a1 - built 12:57:43, Sep 14 2004

Platform: IQ80321 (XScale)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

RAM: 0x00000000-0x08000000, [0x00059668-0x07dd1000] available
FLASH: 0xf0000000 - 0xf0800000, 64 blocks of 0x00020000 bytes each.
RedBoot>
```

Now, the ROM image can be downloaded using the following RedBoot command:

```
RedBoot> load -r -b %{FREEMEMLO} -m ymodem
```

Use the terminal emulator's Ymodem support to send the file `redboot_ROM.bin`. This should result in something like the following output:

```
Raw file loaded 0x0005a000-0x000819f8, assumed entry at 0x0005a000
xyzModem - CRC mode, 911(SOH)/0(STX)/0(CAN) packets, 4 retries
RedBoot>
```

Once the file has been uploaded, you can check that it has been transferred correctly using the **cksum** command. On the host (Linux or Cygwin) run the **cksum** program on the binary file:

```
$ cksum redboot_ROM.bin
140216855 116332 redboot_ROM.bin
```

In RedBoot, run the **cksum** command on the data that has just been loaded:

```
RedBoot> cksum -b %{FREEMEMLO} -l 116332
POSIX cksum = 140216855 116332 (0x085b8a17 0x0001c66c)
```

The second number in the output of the host **cksum** program is the file size, which should be used as the argument to the `-l` option in the RedBoot **cksum** command. The first numbers in each instance are the checksums, which should be equal.

If the program has downloaded successfully, then it can be programmed into the flash using the following commands:

```
RedBoot> fis init
About to initialize [format] FLASH image system - continue (y/n)?y
*** Initialize FLASH Image System
```

```

... Unlock from 0xf07e0000-0xf0800000: .
... Erase from 0xf07e0000-0xf0800000: .
... Program from 0x07cc0000-0x07d00000 at 0xf07e0000: .
... Lock from 0xf07e0000-0xf0800000: .
RedBoot> fis create -b %{\FREEMEMLO} RedBoot
An image named 'RedBoot' exists - continue (y/n)?y
... Unlock from 0xf0000000-0xf0040000: ..
... Erase from 0xf0000000-0xf0040000: ..
... Program from 0x0001c000-0x0005c000 at 0xf0000000: .
... Lock from 0xf0000000-0xf0040000: .
... Unlock from 0xf07e0000-0xf0800000: .
... Erase from 0xf07e0000-0xf0800000: .
... Program from 0x07cc0000-0x07d00000 at 0xf07e0000: .
... Lock from 0xf07e0000-0xf0800000: .
RedBoot>

```

The IQ80321 board may now be reset either by cycling the power, or with the **reset** command. It should then display the startup screen for the ROM version of RedBoot:

```
+No network interfaces found
```

```

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version v2_0_24a1 - built 12:59:55, Sep 14 2004

```

```

Platform: IQ80321 (XScale)
Copyright (C) 2000, 2001, 2002, Free Software Foundation, Inc.
Copyright (C) 2003, 2004, eCosCentric Limited

```

```

RAM: 0x00000000-0x08000000, [0x0001b068-0x07fd1000] available
FLASH: 0xf0000000 - 0xf0800000, 64 blocks of 0x00020000 bytes each.
RedBoot>

```

LED Codes

RedBoot uses the two digit LED display to indicate status during board initialization. Possible codes are:

LED Actions

```

-----
Power-On/Reset
88
  Set the CPSR
  Enable coprocessor access
  Drain write and fill buffer
  Setup PBIU chip selects
A1
  Enable the Icache
A2
  Move FLASH chip select from 0x0 to 0xF0000000
  Jump to new FLASH location
A3

```

```

Setup and enable the MMU
A4
I2C interface initialization
90
Wait for I2C initialization to complete
91
Send address (via I2C) to the DIMM
92
Wait for transmit complete
93
Read SDRAM PD data from DIMM
94
Read remainder of EEPROM data.
An error will result in one of the following
error codes on the LEDs:
77 BAD EEPROM checksum
55 I2C protocol error
FF bank size error
A5
Setup DDR memory interface
A6
Enable branch target buffer
Drain the write & fill buffers
Flush Icache, Dcache and BTB
Flush instruction and data TLBs
Drain the write & fill buffers
SL
ECC Scrub Loop
SE
A7
Clean, drain, flush the main Dcache
A8
Clean, drain, flush the mini Dcache
Flush Dcache
Drain the write & fill buffers
A9
Enable ECC
AA
Save SDRAM size
Move MMU tables into RAM
AB
Clean, drain, flush the main Dcache
Clean, drain, flush the mini Dcache
Drain the write & fill buffers
AC
Set the TTB register to DRAM mmu_table
AD
Set mode to IRQ mode

```

- A7
Move SWI & Undefined "vectors" to RAM (at 0x0)
- A6
Switch to supervisor mode
- A5
Move remaining "vectors" to RAM (at 0x0)
- A4
Copy DATA to RAM
Initialize interrupt exception environment
Initialize stack
Clear BSS section
- A3
Call platform specific hardware initialization
- A2
Run through static constructors
- A1
Start up the eCos kernel or RedBoot

Special RedBoot Commands

A special RedBoot command, **diag**, is used to access a set of hardware diagnostics. To access the diagnostic menu, enter **diag** at the RedBoot prompt:

```
RedBoot> diag
Entering Hardware Diagnostics - Disabling Data Cache!

      IQ80321 Hardware Tests

1 - Memory Tests
2 - Repeating Memory Tests
3 - Repeat-On-Fail Memory Tests
4 - Rotary Switch S1 Test
5 - 7 Segment LED Tests
6 - i82544 Ethernet Configuration
7 - Battery Status Test
8 - Battery Backup SDRAM Memory Test
9 - Timer Test
10 - PCI Bus test
11 - CPU Cache Loop (No Return)
    0 - quit
Enter the menu item number (0 to quit):
```

Tests for various hardware subsystems are provided, and some tests require special hardware in order to execute normally. The Ethernet Configuration item may be used to set the board ethernet address.

Memory Tests

This test is used to test installed DDR SDRAM memory. Five different tests are run over the given address ranges. If errors are encountered, the test is aborted and information about the failure is printed. When selected, the user will be prompted to enter the base address of the test range and its size. The numbers must be in hex with no leading “0x”

```
Enter the menu item number (0 to quit): 1
```

```
Base address of memory to test (in hex): 100000
```

```
Size of memory to test (in hex): 200000
```

```
Testing memory from 0x00100000 to 0x002fffff.
```

```
Walking 1's test:
```

```
00000001000000002000000004000000008000000010000000020000000040000000080
00000100000000200000000400000000800000001000000002000000004000000008000
00010000000020000000040000000080000000100000000200000000400000000800000
01000000002000000004000000008000000010000000020000000040000000080000000
```

```
passed
```

```
32-bit address test: passed
```

```
32-bit address bar test: passed
```

```
8-bit address test: passed
```

```
Byte address bar test: passed
```

```
Memory test done.
```

Repeating Memory Tests

The repeating memory tests are exactly the same as the above memory tests, except that the tests are automatically rerun after completion. The only way out of this test is to reset the board.

Repeat-On-Fail Memory Tests

This is similar to the repeating memory tests except that when an error is found, the failing test continuously retries on the failing address.

Rotary Switch S1 Test

This tests the operation of the sixteen position rotary switch. When run, this test will display the current position of the rotary switch on the LED display. Slowly dial through each position and confirm reading on LED.

7 Segment LED Tests

This tests the operation of the seven segment displays. When run, each LED cycles through 0 through F and a decimal point.

i82544 Ethernet Configuration

This test initializes the ethernet controller's serial EEPROM if the current contents are invalid. In any case, this test will also allow the user to enter a six byte ethernet MAC address into the serial EEPROM.

Enter the menu item number (0 to quit): **6**

Current MAC address: 00:80:4d:46:00:02

Enter desired MAC address: **00:80:4d:46:00:01**

Writing to the Serial EEPROM... Done

***** Reset The Board To Have Changes Take Effect *****

Battery Status Test

This tests the current status of the battery. First, the test checks to see if the battery is installed and reports that finding. If the battery is installed, the test further determines whether the battery status is one or more of the following:

- Battery is charging.
- Battery is fully discharged.
- Battery voltage measures within normal operating range.

Battery Backup SDRAM Memory Test

This tests the battery backup of SDRAM memory. This test is a three step process:

1. Select Battery backup test from main diag menu, then write data to SDRAM.
2. Turn off power for 60 seconds, then repower the board.
3. Select Battery backup test from main diag menu, then check data that was written in step 1.

Timer Test

This tests the internal timer by printing a number of dots at one second intervals.

PCI Bus Test

This tests the secondary PCI-X bus and socket. This test requires that an IQ80310 board be plugged into the secondary slot of the IOP80321 board. The test assumes at least 32MB of installed memory on the IQ80310. That memory is mapped into the IOP80321 address space and the memory tests are run on that memory.

CPU Cache Loop

This test puts the CPU into a tight loop run entirely from the ICache. This should prevent all external bus accesses.

Rebuilding RedBoot

Should it prove necessary to rebuild a RedBoot binary, this is done most conveniently at the command line. The steps needed to rebuild the the ROM version of RedBoot for the IQ80321 are:

```
$ mkdir redboot_iq80321_rom
$ cd redboot_iq80321_rom
$ ecosconfig new iq80321 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/iq80321/VERSION/misc/redboot_ROM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

To rebuild the RAM RedBoot:

```
$ mkdir redboot_iq80321_ram
$ cd redboot_iq80321_ram
$ ecosconfig new iq80321 redboot
$ ecosconfig import $ECOS_REPOSITORY/hal/arm/arm9/iq80321/VERSION/misc/redboot_RAM.ecm
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

At the end of the build the `install/bin` subdirectory should contain the file `redboot.bin`. This is the case for both the above builds, take care not to mix the two files up, since programming the RAM RedBoot into the ROM will render the board unbootable.

Configuration

Name

Configuration — Platform-specific Configuration Options

Overview

The IQ80321 platform HAL package is loaded automatically when eCos is configured for a `iq80321` target. It should never be necessary to load this package explicitly. Unloading the package should only happen as a side effect of switching target hardware.

Startup

The IQ80321 platform HAL package supports two separate startup types:

RAM

This is the startup type which is normally used during application development. The board has RedBoot programmed into flash and boots into that initially. `arm-elf-gdb` is then used to load a RAM startup application into memory and debug it. It is assumed that the hardware has already been initialized by RedBoot. By default, the application will use the eCos virtual vectors mechanism to obtain certain services from RedBoot, including diagnostic output.

ROM

This startup type can be used for finished applications which will be programmed into flash at physical address `0xF0000000`. The application will be self-contained with no dependencies on services provided by other software. eCos startup code will perform all necessary hardware initialization.

RedBoot and Virtual Vectors

If the application is intended to act as a ROM monitor, providing services for other applications, then the configuration option `CYGSEM_HAL_ROM_MONITOR` should be set. Typically this option is set only when building RedBoot.

If the application is required to make use of services provided by a ROM monitor, via the eCos virtual vector mechanism, then the configuration option `CYGSEM_HAL_USE_ROM_MONITOR` should be set. By default this option is enabled when building for RAM startup, disabled otherwise. It can be manually disabled for RAM startup, making the application self-contained, as a testing step before switching to ROM startup.

If the application does not rely on a ROM monitor for diagnostic services then the serial port will be claimed for HAL diagnostics.

Flash Driver

The IQ80321 board contains an 8MB Intel StrataFlash flash device. The `CYGPKG_DEVS_FLASH_STRATA` package contains all the code necessary to support these parts and the `CYGPKG_DEVS_FLASH_IQ80321` package contains definitions that customize the driver to the IQ80321 board.

Ethernet Driver

The IQ80321 board contains an Intel i82544 ethernet controller. The `CYGPKG_DEVS_ETH_INTEL_I82544` package contains all the code necessary to support this device and the `CYGPKG_DEVS_ETH_ARM_IQ80321` package contains definitions that customize the driver to the IQ80321 board.

System Clock

By default, the system clock interrupts once every 10ms, corresponding to a 100Hz clock. This can be changed by the configuration option `CYGNUM_HAL_RTC_DENOMINATOR` which corresponds to the clock frequency. Other clock-related settings are recalculated automatically if the denominator is changed.

Compiler Flags

The platform HAL defines the default compiler and linker flags for all packages, although it is possible to override these on a per-package basis. Most of the flags used are the same as for other architectures supported by eCos. There is just one flag specific to this port:

`-mcpu=xscale`

The arm-elf-gcc compiler supports many variants of the ARM architecture. A `-m` option should be used to select the specific variant in use, and with current tools `-mcpu=xscale` is the correct option for the XScale CPU on the IOP321.

The HAL Port

Name

HAL Port — Implementation Details

Overview

This documentation explains how the eCos HAL specification has been mapped onto the IQ80321 hardware, and should be read in conjunction with that specification. The IQ80321 platform HAL package complements the ARM architectural HAL, the XScale core HAL and the IOP321 (VERDE) variant HAL. It provides functionality which is specific to the target board.

Startup

Following a hard or soft reset the HAL will initialize or reinitialize the on-chip peripherals that are used by eCos. There is an exception for RAM startup applications which depend on a ROM monitor for certain services.

For ROM startup, the HAL will perform additional initialization, setting up the external SDRAM and programming the various internal registers. This is all done in the `PLATFORM_SETUP1` macro in the assembler header file `hal_platform_setup.h`.

Memory Maps

The RAM based page table is located at RAM start + 0x4000.

NOTE: The virtual memory map in this section uses a C, B, and X column to indicate the caching policy for the region.

X	C	B	Description
-	-	-	-----
0	0	0	Uncached/Unbuffered
0	0	1	Uncached/Buffered
0	1	0	Cached/Buffered Write Through, Read Allocate
0	1	1	Cached/Buffered Write Back, Read Allocate
1	0	0	Invalid -- not used
1	0	1	Uncached/Buffered No write buffer coalescing
1	1	0	Mini DCache - Policy set by Aux Ctl Register
1	1	1	Cached/Buffered Write Back, Read/Write Allocate

Physical Address Range	Description
-----	-----
0x00000000 - 0x7fffffff	ATU Outbound Direct Window
0x80000000 - 0x900fffff	ATU Outbound Translate Windows
0xa0000000 - 0xbfffffff	SDRAM
0xf0000000 - 0xf0800000	FLASH (PBIU CS0)

0xfe800000 - 0xfe800fff	UART	(PBIU CS1)
0xfe840000 - 0xfe840fff	Left 7-segment LED	(PBIU CS3)
0xfe850000 - 0xfe850fff	Right 7-segment LED	(PBIU CS2)
0xfe8d0000 - 0xfe8d0fff	Rotary Switch	(PBIU CS4)
0xfe8f0000 - 0xfe8f0fff	Battery Status	(PBIU CS5)
0xfff00000 - 0xffffffff	Verde Memory mapped Registers	

Default Virtual Map	X	C	B	Description
-----	-	-	-	-----
0x00000000 - 0x1fffffffff	1	1	1	SDRAM
0x20000000 - 0x9fffffffff	0	0	0	ATU Outbound Direct Window
0xa0000000 - 0xb00fffffffff	0	0	0	ATU Outbound Translate Windows
0xc0000000 - 0xdfffffffff	0	0	0	Uncached alias for SDRAM
0xe0000000 - 0xe00fffffffff	1	1	1	Cache flush region (no phys mem)
0xf0000000 - 0xf0800000	0	1	0	FLASH (PBIU CS0)
0xfe800000 - 0xfe800fff	0	0	0	UART (PBIU CS1)
0xfe840000 - 0xfe840fff	0	0	0	Left 7-segment LED (PBIU CS3)
0xfe850000 - 0xfe850fff	0	0	0	Right 7-segment LED (PBIU CS2)
0xfe8d0000 - 0xfe8d0fff	0	0	0	Rotary Switch (PBIU CS4)
0xfe8f0000 - 0xfe8f0fff	0	0	0	Battery Status (PBIU CS5)
0xfff00000 - 0xffffffff	0	0	0	Verde Memory mapped Registers

Other Issues

The IQ80321 platform HAL does not affect the implementation of other parts of the eCos HAL specification. The XScale core HAL, the IOP321 (VERDE) variant HAL, and the ARM architectural HAL documentation should be consulted for further details.

XXXVIII. FAT File System Support

Chapter 12. Introduction

This document describes the FAT filesystem provided in eCos. This is implemented by the FATFS package which uses the facilities of the FILEIO package to present its functionality to the user.

The FAT filesystem supports FAT12, FAT16 and FAT32 disk formats and Long Filename Support is available as a configuration option.

Chapter 13. Configuring the FAT Filesystem

This chapter shows how to include the FAT filesystem into an eCos configuration and how to configure it once installed.

Including FAT Filesystem in a Configuration

The FAT filesystem is contained in a single eCos package, CYGPKG_FS_FAT. However, it depends on the services of a collection of other packages for complete functionality:

CYGPKG_IO_FILEIO

The File IO package. This provides the POSIX compatible API by which the FAT filesystem is accessed.

CYGPKG_IO

Device IO package. This provides all the infrastructure for the disk devices.

CYGPKG_IO_DISK

Disk device IO support. This provides the top level generic disk driver functions. It also interprets partition tables and provides a separate access channel for each partition. This package is described in detail elsewhere.

CYGPKG_LINUX_COMPAT

Linux compatibility library. The FAT filesystem only used the list and RBtree features of this library.

CYGPKG_LIBC_STRING

Strings library. This provides the string and memory move and compare routines used by the filesystem.

CYGPKG_MEMALLOC

The FAT filesystem currently uses malloc() to allocate its memory resources, such as the node and block caches, so this package is required.

To add the FAT filesystem to a configuration, it is necessary to add all of these packages. This is best done by using an import file. The following file will add the FAT filesystem and all the necessary packages to any configuration:

```
cdl_savefile_version 1;
cdl_savefile_command cdl_savefile_version {};
cdl_savefile_command cdl_savefile_command {};
cdl_savefile_command cdl_configuration { description hardware template package };
cdl_savefile_command cdl_package { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_component { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_option { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_interface { value_source user_value wizard_value inferred_value };

cdl_configuration eCos {
```

```
package CYGPKG_FS_FAT current ;
package CYGPKG_IO_DISK current ;
package CYGPKG_LINUX_COMPAT current ;
package CYGPKG_IO_FILEIO current ;
package CYGPKG_IO current ;
package CYGPKG_LIBC_STRING current ;
package CYGPKG_MEMALLOC current ;
};
```

In addition to these packages, hardware-specific device driver packages will be needed for the disk devices to be used. These device drivers are usually part of the target description in the eCos database and will be enabled if the CYGPKG_IO_DISK package is included.

Configuring the FAT Filesystem

Once added to the configuration, the FAT filesystem has a number of configuration options:

CYGNUM_FS_FAT_NODE_HASH_TABLE_SIZE

This option controls the number of slots in the hash table used to store file nodes using filenames as keys.

Default value: 32

CYGNUM_FS_FAT_NODE_POOL_SIZE

This option controls the size of the node pool used for storing file nodes. This value should be set to the maximum required number of simultaneously open files plus the desired size of unused node cache.

Default value: CYGNUM_FILEIO_NFILE + 2

CYGNUM_FS_FAT_BLOCK_CACHE_BLOCKSIZE

This option controls the size of blocks in the block cache. This value should be a power-of-2 multiple of 512.

Default value: 512

CYGNUM_FS_FAT_BLOCK_CACHE_MEMSIZE

This option controls the amount of memory used for the block cache.

Default value: 20 * CYGNUM_FS_FAT_BLOCK_CACHE_BLOCKSIZE

CYGDBG_FS_FAT_NODE_CACHE_EXTRA_CHECKS

This option controls the inclusion of extra sanity checks in node cache code.

Default value: 1

`CYGCFG_FS_FAT_USE_ATTRIBUTES`

This option controls whether the FAT filesystem supports or honors the FAT filesystem file attributes.

Default value: 0

`CYGCFG_FS_FAT_LONG_FILE_NAMES`

This option controls the FAT filesystem support for long file names.

Default value: 0

`CYGNUM_FS_FAT_LONG_FILE_NAME_MAX`

This option defines the maximum size of long file names supported by the filesystem. The default value of 64 corresponds to `NAME_MAX`, which defines the size of `d_name[]` in a struct `dirent`.

Default value: 64

Normally these options should be left as they are unless you have a specific need to change them. Once the configuration had been created, it should be possible to compile eCos and link it with the application without any errors.

Chapter 14. Using the FAT Filesystem

The FAT filesystem is accessed through the FILEIO package and responds to all the standard filesystem functions such as `open()`, `close()`, `read()` and `write()`. To use these operations the filesystem must first be mounted.

A FAT filesystem may be mounted using the `mount()` function. The following is an example of how to mount a FAT filesystem:

```
err = mount( "/dev/hd0/1", "/disk0", "fatfs:sync=write" );
```

This function call will mount the first partition of hard disk 0 (see the documentation on the DISKIO package for a full description of the device name format). The root of this disk can then be accessed with the name `"/disk0"`. The `mount()` function will return zero if the mount succeeded, or -1 if it failed for any reason, for example if the partition does not exist, or the filesystem is not in FAT format.

The options after the colon in the filesystem name are passed to the filesystem to control various aspects of the filesystem. The options currently supported are:

sync

This option controls the synchronization behaviour of the block cache. If omitted then the cache is run on an entirely write-back basis and blocks are only written back to disk when they need to be replaced with new data, when `sync()` is called, or the filesystem is unmounted. This is generally the most efficient mode, but is prone to losing data or corrupting the filesystem if power is lost while the filesystem is mounted.

If this option is set to "write" then the cache is operated on a write-back basis and every block update is written immediately back to disk. This is the least efficient mode since any extension to a file may result in several blocks being written back to disk. It does, however, keep the filesystem up to date on disk.

If this option is set to "close" then the cache is only written back to disk whenever a file is closed. Note that this causes the entire cache to be written, not just those blocks associated with the file being closed. In terms of efficiency, this is a good compromise between performance and safety.

readonly

This is a stand-alone option which causes the filesystem to be mounted read-only. The effect of this is to prevent the filesystem writing anything back to the disk. Under normal circumstances this cannot be guaranteed for a normal mount, even when files are only read, since the filesystem may need to update the access time for files that have been read.

When finished with, a filesystem may be unmounted using the `umount()` function. The following would unmount the filesystem mounted above:

```
err = umount( "/disk0" );
```

Warning!

It is important to unmount any removable devices before removing them, otherwise there is no guarantee that all cached data blocks will have been written to disk. The same is true of resetting the system before unmounting non-removable devices.

Chapter 15. Removable Media Support

The FAT filesystem has support for Removable Media, which is implemented in conjunction with support in the FILEIO package, the generic DISKIO layer and the hardware disk driver. At present, only the JUNG0 USB mass storage device contains the necessary support.

To support Removable media, application code, or the automounter, should register a disk event callback to capture device insertion and removal. When an insert event is detected, an attempt to mount the filesystem should be made. If this is successful, then the files on the device can be accessed.

When the device is removed the hardware driver will reject all IO operations with an EIO response. The filesystem will propagate these errors back to any current file IO operations which will result in `read()` or `write()` returning an EIO error. Application code should be ready for this to happen.

A disk event callback will also be delivered and the application should arrange to call `umount_force()` to force the filesystem to be unmounted. The FAT filesystem handles this by releasing all resources and detaching from the disk device.

To help applications indicate to the user whether the medium may be removed (by means of a LED or an on-screen icon) the FAT filesystem supports the filesystem callback mechanism defined in the FILEIO package. This relies on the following definitions in `fileio.h`:

```
typedef void cyg_fs_callback( cyg_int32 event, CYG_ADDRWORD data );

struct cyg_fs_callback_info
{
    cyg_fs_callback    *callback;        // Callback function
    CYG_ADDRWORD        data;            // User data value
};

// Callback events
#define CYG_FS_CALLBACK_SAFE    1        // The filesystem is up to date on disk
#define CYG_FS_CALLBACK_UNSAFE  2        // The filesystem is not up to date
```

A callback function may be registered after a filesystem has been mounted by using `cyg_fs_setinfo()` as follows:

```
struct cyg_fs_callback_info callback;

...

callback.callback = fs_callback;
callback.data = my_data;
err = cyg_fs_setinfo("/disk0", FS_INFO_CALLBACK, &callback, sizeof(callback));
```

Following this, whenever the filesystem has dirty cache blocks that are not up to date on the disk, `fs_callback()` will be called with `event` set to `CYG_FS_CALLBACK_UNSAFE`. When all blocks become up to date on disk it will be called with `CYG_FS_CALLBACK_SAFE`.

Chapter 16. Non-ASCII Character Set Support

If long filename support is enabled (by setting `CYGCFG_FS_FAT_LONG_FILE_NAMES`) then all strings passed to and from the filesystem may be encoded using UTF-8. This allows files to be named using characters beyond the basic ASCII set.

If long filename support is disabled then file names are limited to the standard 8.3 format. However, these file names are preserved in an 8-bit clean format, if they contain non-ASCII characters, so that any multi-byte encodings are preserved.

Filesystems created by devices that do not support long filenames may have 8.3 names that are encoded using non-ASCII and non-Unicode character sets. Typically these will be encoded according to Microsoft code page character sets. To permit these names to pass through the rest of the filesystem, and compare correctly during file searches, when long filename support is enabled, these names need to be translated into Unicode. Since the filesystem has no built-in internationalization support, beyond Unicode, it is the responsibility of the application or middleware layers to supply the translation of these values to and from Unicode. The FILEIO package defines callbacks that may be used to do this:

```
typedef int cyg_fs_mbcstoutf16le( CYG_ADDRWORD data, const cyg_uint8 *mbcs, int size, cyg_uint16 *utf16le);
typedef int cyg_fs_utf16leto_mbcst( CYG_ADDRWORD data, const cyg_uint16 *utf16le, int size, cyg_uint8 *mbcs);

struct cyg_fs_mbcsttranslate
{
    cyg_fs_mbcstoutf16le    *mbcstoutf16le;
    cyg_fs_utf16leto_mbcst *utf16leto_mbcst;
    CYG_ADDRWORD            data;
};
```

These callback functions may be registered after a filesystem has been mounted by using `cyg_fs_setinfo()` as follows:

```
struct cyg_fs_mbcsttranslate translate;

...

translate.mbcstoutf16le = my_mbcstoutf16le;
translate.utf16leto_mbcst = my_utf16leto_mbcst;
translate.data = (CYG_ADDRWORD)my_data;
err = cyg_fs_setinfo("/disk0", FS_INFO_MBCSTTRANSLATE, &translate, sizeof(translate));
```

Following this, whenever the filesystem encounters a short file name that contains non-ASCII characters the registered `mbcstoutf16le()` function will be called to translate it. In the call, the *data* argument will be a copy of the *data* field of the `cyg_fs_mbcsttranslate` structure. The *mbcs* argument points to the sequence of *size* bytes to be translated. The resulting translation should be stored in *utf16le* and the number of 16-bit values stored returned from the function.

When the filesystem needs to encode a string into the multibyte character set, it will call the `utf16leto_mbcst()` function. In the call, the *data* argument will be a copy of the *data* field of the `cyg_fs_mbcsttranslate` structure.

The *utf16le* argument points to the sequence of *size* 16-bit values to be translated. The resulting translation should be stored in *mbcs* and the number of bytes stored returned from the function.

It is important to note that translation is to and from UTF-16LE. All 16 bit values are stored in little endian byte order and Unicode code points outside the Basic Multilingual Plane are encoded as surrogate pairs. This is the format mandated by Microsoft for long file names in the FAT filesystem. See IETF RFC2781 for details of the encoding.

In the current implementation the `utf16le_to_mbc()` will not be called. If long filename support is disabled, then the filesystem will store multibyte characters as they are supplied. If long filename support is enabled then new files will be created with long names if any non-ASCII characters are present. Renamed files will be converted to the long name form automatically. This function is present in case future enhancements require it. For now applications should install a function that simply returns zero.

Chapter 17. Testing

There are currently four tests available for the FAT filesystem, **fatfs1**, **fatfs2** and **fatfs3**. **fatfs1** is a simple functional test of the filesystem, ensuring that files and directories can be created, renamed and deleted, and that various error conditions can be detected. **fatfs2** tests the filesystem's ability to handle multi-threaded access by creating several threads that access the filesystem in parallel. **fatfs3** is designed to test support for changing disks. **fatfs4** tests support for long file names and is only built if this feature is enabled.

Testing the FAT filesystem depends on the availability of a suitable device to perform the tests on. The following configuration options are defined in the target specific disk device driver to configure the tests for the available hardware:

CYGDAT_DEVS_DISK_TEST_DEVICE

Device name of test disk or partition. This device will be mounted on the mountpoint given in `CYGDAT_DEVS_DISK_TEST_MOUNTPOINT` and tests carried out in the directory given by `CYGDAT_DEVS_DISK_TEST_DIRECTORY`.

CYGDAT_DEVS_DISK_TEST_MOUNTPOINT

Mountpoint for test disk.

CYGDAT_DEVS_DISK_TEST_DIRECTORY

Subdirectory in test device where tests can create files and directories.

CYGDAT_DEVS_DISK_TEST_DEVICE2

Device name of optional second test disk or partition. If this is not defined then the tests will carry out any operations that would have been executed on the second disk in the test directory on the main test disk.

CYGDAT_DEVS_DISK_TEST_MOUNTPOINT2

Mountpoint for optional second test disk. If `CYGDAT_DEVS_DISK_TEST_DEVICE2` is not defined then this option is not needed.

CYGDAT_DEVS_DISK_TEST_DIRECTORY2

Subdirectory in optional second test device where tests can create files and directories. If `CYGDAT_DEVS_DISK_TEST_DEVICE2` is not defined then this option is not needed.

XXXIX. Journalling Flash File System v2 (JFFS2)

Journalling Flash File System v2 overview

Name

CYGPKG_FS_JFFS2 — Provides Journalling file system for Flash

Description

The Journalling Flash File System version 2 (JFFS2) provides a robust file system to allow reliable use of Flash devices as data storage. The eCos implementation is greatly shared with the Linux kernel implementation, thus ensuring compatibility and encouraging development.

JFFS2 was designed from the outset for embedded devices. It allows recovery when the system has failed abnormally, without the file system itself being left in an unusable state, even if power is disconnected at the moment the Flash device is in the middle of being written to. It also offers features such as compression for efficient data storage, and garbage collection to improve capacity. Most importantly it is fully integrated into the eCos file I/O infrastructure as a plug-in filesystem.

External references

There are a number of external resources containing information about JFFS2 on the internet, other than the usual eCos-specific general resources. The key site is the Linux MTD website (<http://www.linux-mtd.infradead.org/>) which, although clearly having a Linux focus, contains lots of useful documentation on JFFS2 as well as a mailing list with searchable archives. The mailing list welcomes questions on using JFFS2 on eCos. Note that the eCos JFFS2 port does not use the MTD layer itself.

Another useful site is the Red Hat JFFS2 (<http://sources.redhat.com/jffs2/>) website, which contains a very useful paper presented to a Linux symposium covering the internals and some of the design of JFFS2.

Using JFFS2

Name

JFFS2 usage — Description of how to use JFFS2

Mounting

A JFFS2 filesystem can be mounted just like any normal eCos filesystem, using the `mount()` function from the POSIX file I/O package (`CYGPKG_FILEIO`). You must choose an appropriate Flash I/O block device to use. Documentation on Flash I/O block devices can be found in the Generic Flash package documentation.

Example 1. Mounting and unmounting a JFFS2 filesystem

```
#include <cyg/fileio/fileio.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

...
int rc;
rc = mount( "/dev/flash/fis/jffs2test", "/fs", "jffs2" );
if (rc < 0)
    printf( "mount returned error: %s\n", strerror(errno) );
...
rc = umount( "/fs" );
if (rc < 0)
    printf( "umount returned error: %s\n", strerror(errno) );
...
```

No file system needs to be created in advance for JFFS2. A new file system image will be instantiated if JFFS2 is pointed at an erased Flash area. Similarly if JFFS2 is pointed at a non-erased Flash area that does not contain valid JFFS2 markers, it will refuse to mount it to prevent destruction of data.

If mounting an existing JFFS2 filesystem, the mount procedure will search for any unused blocks that have not already been erased, and erase them. This can result in an extended mount time, and so this feature can be disabled with the CDL option `CYGOPT_FS_JFFS2_ERASE_PENDING_ON_MOUNT`.

Normally it is sufficient to prepare a clean JFFS2 partition as above, and load files into it using RedBoot or an eCos application. But if you do wish to use a pregenerated file system image generated on your host PC, there is a utility named **mkfs.jffs2** which can be used to generate an image. Be sure to use the `-l` or `-b` options to select the endianness so it corresponds with your target, and you may need to set other options according to the requirements of your Flash hardware such as erase block size - use the `--help` option for a list of parameters.

Note that JFFS2's memory requirements are not static, and so they may increase over time before stabilising. Larger Flash partitions may require non-trivial amounts of memory, especially at mount time. Memory use may be controlled by removing features such as compression, or by constraining the size of the Flash partition. Configuration options controlling optional features may be found in the JFFS2 package CDL configuration.

JFFS2 has built-in tolerance of Flash errors when erasing and should adapt and work around bit errors that arise as Flash reaches the end of its working life. Obviously this comes at the expense of device capacity.

Garbage collection

By default, JFFS2 performs garbage collection on an as-needed basis. This means that when there are insufficient spare clean flash blocks remaining, JFFS2 will perform repeated garbage collection until a block can be erased and then used for future writes.

Garbage collection involves scanning a flash block and determining which nodes are still used for valid data and which are obsolete - valid data is then relocated to a new alternative block, until only obsolete data remains, at which point the block can be erased. The algorithms which choose which block is nominated for garbage collection do so to ensure wear levelling over the life of the flash device. Nodes representing individual file fragments are able to be coalesced and merged with adjacent file fragments leading to reduced flash use overall due to eliminating some of the overhead caused by node metadata on the Flash. There will also be a consequent simplification in the internal filesystem data structures, resulting in reduced memory consumption and fewer data structures which JFFS2 needs to trawl through when locating data. It will also greatly augment the benefit of compression: when compressing, nodes are considered in isolation, and so small nodes are unlikely to compress well, whereas larger coalesced nodes are more likely to occupy less flash space.

Garbage collection thread

As a result of only performing garbage collection when needed, it can mean that individual writes to files may occasionally take a lengthy period of time to run if a new flash block is required - a whole flash block may need to be garbage collected from scratch, have its live data written to a new block, and be erased, the latter in particular being a very lengthy procedure. As such, JFFS2 offers the option of using a *garbage collection thread*, which can run in the background to advance the garbage collection process when the filesystem is not otherwise being used.

Of course if a filesystem is going to change too rapidly, or the application is CPU bound by higher priority threads, then the garbage collection thread may not be able to keep up. But for the majority of applications, it can considerably reduce or even virtually eliminate the delays caused by the requirement for occasional garbage collection.

The garbage collection thread can be enabled in the JFFS2 package's configuration using the "Garbage collection background thread" CDL option (`CYGOPT_FS_JFFS2_GC_THREAD`). The priority of that thread defaults to the lowest possible - 1 above that of the idle thread, but this can be changed with the `CYGNUM_JFFS2_GC_THREAD_PRIORITY` option. The thread of course requires a stack for execution, the value of which can be optimised with the `CYGNUM_JFFS2_GC_THREAD_STACK_SIZE` option. Note that one garbage collection thread is started for each mounted JFFS2 filesystem, and so one thread stack is allocated for each.

Usually the garbage collection thread will gradually garbage collect nodes from erase blocks until the block is completely unused by valid data. However it would not actually erase it, leaving that to the usual code path that erases blocks only at the point they are needed. This is because otherwise the Flash system may get locked for the duration of the erase process, preventing any threads, including high priority threads, access to it at that time. Yet the garbage collection thread is intended to be a low priority background thread. Nevertheless, enabling the option `CYGSEM_JFFS2_GC_THREAD_CAN_ERASE` allows the garbage collection thread to erase blocks as well, if blocks are available for erasure.

Finally, the garbage collection thread may run continuously but does not have to run constantly - the number of ticks between garbage collection passes can be specified with the `CYGNUM_JFFS2_GC_THREAD_TICKS` configuration option. How this corresponds to real time is unspecified and depends on the HAL and kernel clock configuration (although most ports have traditionally defaulted to 10ms ticks). The value of this option can even be set to 0. Note though, that garbage collection is never considered "done" - the thread will run continuously until the filesystem is unmounted, therefore making it run too frequently may in fact cause unnecessary flash operations, increasing flash wear (even though it will be levelled wear). The value should be chosen according to the expected write pattern.

The number of ticks between garbage collection passes can also be set at runtime, by invoking a `cyg_fs_setinfo()` call on a filesystem object. There is a corresponding `cyg_fs_getinfo()` call to retrieve the current delay ticks. For example:

```
#include <cyg/fs/jffs2/jffs2.h>

...
{
    int err;
    cyg_tick_count_t old_delay_ticks;
    cyg_tick_count_t new_delay_ticks;

    err = cyg_fs_getinfo("/jffs2", FS_INFO_JFFS2_GET_GC_THREAD_TICKS, &old_delay_ticks, sizeof(cy
    assert(err == 0);
    new_delay_ticks = old_delay_ticks*10; // slow down GC
    err = cyg_fs_setinfo("/jffs2", FS_INFO_JFFS2_SET_GC_THREAD_TICKS, &new_delay_ticks, sizeof(cy
    assert(err == 0);
}
```

One application of dynamically setting the wakeup delay for the garbage collection thread is so that the thread is more active in times of relative system activity, but operates more slowly when quiescent. This may improve flash life, while still retaining benefits of the garbage collection thread.

Efficiency

JFFS2 does not guarantee 100% optimal use of Flash space due to its journalling nature, and the granularity of Flash blocks. It is possible for it to fill up even when not all file space appears to have been used, especially if files have had many small operations performed on them and the Flash partition is small compared to the size of the Flash blocks. It is strongly recommended to have at least 5 or 6 Flash blocks spare, over and above space requirements for data, in order to allow the JFFS2 garbage collector to operate.

It is certainly the case that JFFS2 will work very inefficiently if using many small writes. Unless and until garbage collected, each write will occupy its own JFFS2 node on Flash, and so will incur overhead from the node header. A filesystem is likely to "fill" quickly if written a few bytes at a time, rather than in large chunks, so caution is advised, and more space reserved if that write pattern is anticipated. A common application that can do this is logging. Small writes can also remove the benefits of compression, as there is too little data to compress effectively.

Use of the garbage collection thread will provide a level of continuous garbage collection. As indicated [above](#), garbage collection can reduce flash usage and memory consumption, and improve performance. Therefore doing so on a continuous basis is a wise idea.

Extra spare space is required in order to allow the JFFS2 garbage collector to operate, over and above space requirements for data. A rule of thumb is to use the following formula:

$$\text{Recommended overhead} == 2 + ((\text{flashsize}/50) + (\text{flashsectors} * 100) + (\text{sectorsize} - 1)) / \text{sectorsize}$$

So for example, a 16Mbyte flash with 64Kbyte blocks given over entirely to JFFS2 would actually require an overhead of 6 blocks. Or to look at it another way, trying to write data when you have used up all but 6 blocks worth may result in an ENOSPC error being reported. Due to metadata and write characteristics (e.g. lots of 1 byte

writes) it's not possible to easily calculate what that actually translates to in terms of maximum file size. Even the above formula is only a rule of thumb, and it has not been proven to be guaranteed to work in all circumstances. It is recommended to be conservative if possible.

JFFS2 will default to trying to compress files. However, it may be more memory efficient to disable JFFS2 compression entirely in the CDL configuration, and instead ensure that images are stored compressed when they are downloaded, and use the standard RedBoot mechanism to decompress the files upon loading.

Configuration dependencies

JFFS2 has a number of package dependencies. As such it may be helpful to use the below eCos minimal configuration (.ecm) file and import it into your configuration to satisfy most dependencies quickly without conflict. This minimal configuration file is usable for building both eCos and RedBoot with JFFS2 included. Note you may need to modify the package versions from *current* to the version of your release, e.g. *v2_0_64*.

```
cdl_configuration eCos {
    package CYGPKG_IO_FLASH current ;
    package CYGPKG_MEMALLOC current ;
    package CYGPKG_COMPRESS_ZLIB current ;
    package CYGPKG_IO_FILEIO current ;
    package CYGPKG_FS_JFFS2 current ;
    package CYGPKG_ERROR current ;
    package CYGPKG_LINUX_COMPAT current ;
    package CYGPKG_IO current ;
    package CYGPKG_CRC current ;
    package CYGPKG_LIBC_STRING current ;
};

cdl_option CYGPKG_IO_FILEIO_DEVFS_SUPPORT {
    user_value 1
};

cdl_component CYGPKG_IO_FLASH_BLOCK_DEVICE {
    user_value 1
};
```

For example:

```
$ ecosconfig new adderII
$ ecosconfig import jffs2.ecm
$ ecosconfig tree
$ make tests
```

Use with RedBoot

JFFS2 support can be built into RedBoot using the [above minimal configuration file](#). In most cases, the configuration settings will then make all the adjustments necessary.

However note that a build of RedBoot which includes JFFS2 with RedBoot, is likely to require more Flash space for its own image, as well as much more RAM space to run. The latter is particularly important to note given that this can reduce the size of the program image which can be loaded into RAM from a JFFS2 filesystem.

Particularly large JFFS2 filesystems, or filesystems with a large number of nodes, require more RAM to be used for JFFS2's in-memory data structures. As such, the value of the configuration option controlling the size of the RedBoot heap (`CYGMEM_REDBOOT_WORKSPACE_HEAP_SIZE`) may need to be increased in such cases. JFFS2 will already make the default size of this heap occupy 192KiB of RAM.

Secure Erase

The eCosPro® port of JFFS2 includes a *Secure Erase* feature. This feature allows the application to ensure that when a file is deleted, its contents are fully erased from the flash.

Usually deleted files persist in Flash for an indeterminate period of time, marked as obsolete. The possible solution taken with other filesystems of trying to overwrite the file data before deletion does not work with JFFS2, as JFFS2 will still retain the old file data in Flash, but writes additional nodes with a higher node version number, and rendering the previous data obsolete. Therefore the Secure Erase functionality can be used to guarantee that a deleted file will have its past contents wiped from the Flash.

Methodology

Because obsolete data for a file could exist in any block, the only way to achieve this is to ensure that every block (other than bad or completely free blocks) is wiped, taking care to relocate live data. This effectively means methodically garbage collecting and erasing every flash block used by the filesystem.

Operation time

For a filesystem which almost completely consists of used flash blocks the secure erase process could take a considerable amount of time during which the filesystem cannot be used for other operations. Therefore it is strongly recommended that the [garbage collection thread](#) support is enabled. With the garbage collection thread running, more blocks are likely to be completely clean, or at least partially garbage collected, thus reducing the time for secure erasure.

Usage

Support for the secure erase functionality must first be enabled with a CDL configuration option - `CYGOPT_FS_JFFS2_SECERASE`.

Then a secure erase operation can be performed on the filesystem with a `cyg_fs_setinfo()` function call using the `FS_INFO_SECURE_ERASE` config key. This call can be invoked specifying any file or directory within the filesystem including its mount point, although the operation itself will take place on the entire filesystem.

Example 2. Secure erase usage

```
#include <cyg/fileio/fileio.h>
#include <errno.h>
#include <stdio.h>
```

```
...
int err;
err = cyg_fs_setinfo("/fs", FS_INFO_SECURE_ERASE, NULL, 0);

if (ENOERR != err)
{
    printf( "Secure erase failed: %d\n", strerror(err) );
    ...
}
```

Testing JFFS2

JFFS2 comes with a number of tests that may be run as normal eCos test applications. To run these tests, you should create a FIS partition in RedBoot named “jffs2test”.

Without a FIS partition of this name, you must set the CDL configuration options `CYGNUM_FS_JFFS2_TEST_OFFSET` and `CYGNUM_FS_JFFS2_TEST_LENGTH`. The tests will attempt to use the region identified by that offset/length combination, but will first check it is blank, and will report a test failure if it is not.

When the tests run, they will erase the Flash test area (usually the “jffs2test” FIS partition) in its entirety, so do not use an existing JFFS2 partition in this space.

The tests are designed to test both general features of JFFS2, as well as do a limited stress-test JFFS2 in the presence of multiple threads.

More specifically, the `jffs2-fileio1` test checks a wide variety of file system operations including creating and removing files and directories, scanning directories, and reading and writing file contents. It also repeats to verify that unmounting and remounting works.

The `jffs2-fseek1` test verifies file seek operations on JFFS2 files, using standard I/O C library calls.

Test files with names of the form `jffs2-NtNf` verify operation with varying numbers of threads, and varying numbers of files.

The test `jffs2_3` is specifically to verify operation of the garbage collection code, and performs a small set of operations repeatedly to do so. It also gives an opportunity for the garbage collection thread to be tested, if enabled.

The test `jffs2-secerasel` is specifically to verify operation of the secure erase facility, if the `CYGOPT_FS_JFFS2_SECERASE` CDL configuration option has been enabled. It also provides further testing of the garbage collection code and the garbage collection thread.

XL. Disk IO Package

Chapter 18. Introduction

This document describes the Disk I/O subsystem provided in eCos. This is implemented by the DISKIO package. This package presents a disk driver API to clients, interprets partition tables and provides the infrastructure in which hardware-specific disk device drivers operate.

Chapter 19. Configuring the DISK I/O Package

This chapter shows how to include the DISK I/O package in an eCos configuration and how to configure it once installed.

Including DISK I/O in a Configuration

The DISK I/O subsystem is contained in a single eCos package, `CYGPKG_IO_DISK`. However, it depends on the services of the following other packages for complete functionality:

`CYGPKG_IO`

Device IO package. Disk devices operate within the generic device infrastructure.

`CYGPKG_ERROR`

The error package. Disk devices need to return standard error codes when operations fail.

The DISK I/O package can be added to any configuration just by adding its package during configuration. However, it is usually added as part of a group which also includes a filesystem implementation and hardware device drivers.

Configuring the DISK I/O Package

The option "Detect FAT boot sector" (`CYGSEM_IO_DISK_DETECT_FAT_BOOT`) can be used with certain types of removable media which are treated undeterministically by Windows PCs, such as USB keys. These usually contain a partition table, but if completely wiped, Windows will not recreate the partition table, and instead place a FAT filesystem directly on the device with no partition table. Enabling this option detects this and fabricates a partition table in memory instead. This option is usually only needed on systems which use removable media likely to be shared with Windows PCs, as indicated from low level drivers with the `CYGINT_IO_DISK_DETECT_FAT_BOOT_DEFAULT` CDL interface.

The CDL interface `CYGINT_IO_DISK_ALIGN_BUFS_TO_CACHELINE` indicates to the disk I/O package, and to its users, that data transfer buffers need to be aligned to data cache line boundaries.

The CDL interface `CYGINT_IO_DISK_REMOVABLE_MEDIA_SUPPORT` indicates that the hardware disk driver implements removable media support. This will cause the disk I/O package to implement the `CYG_IO_SET_CONFIG_DISK_EVENT` and `CYG_IO_GET_CONFIG_DISK_EVENT` configuration option keys.

Chapter 20. Usage

The user API for disk devices is the same as that for serial devices except that `cyg_io_bread()` and `cyg_io_bwrite()` are used for data transfers instead of `cyg_io_read()` and `cyg_io_write()`.

```
// Write data to a block device
Cyg_ErrNo cyg_io_bwrite(
    cyg_io_handle_t handle,
    const void *buf,
    cyg_uint32 *len,
    cyg_uint32 pos )
```

This function sends data to a device. The size of data to send is contained in `*len` and the actual size sent will be returned in the same place. This value must be a count of 512 byte sectors. The `pos` specifies the position on the disk to which the data will be written. It is a linear sector number.

```
// Read data from a block device
Cyg_ErrNo cyg_io_bread(
    cyg_io_handle_t handle,
    void *buf,
    cyg_uint32 *len,
    cyg_uint32 pos )
```

This function receives data from a device. The desired size of data to receive is contained in `*len` and the actual size obtained will be returned in the same place. This value must be a count of 512 byte sectors. The `pos` specifies the position on the disk from which the data will be read. It is a linear sector number.

Disk devices are named in the same way as other devices, thus `"/dev/hd0"` would name the first hard disk. The exact names used for any disk are usually part of the configuration for the target-specific device driver.

Disk devices may be partitioned using a standard DOS partition table. If this is the case then an additional element to the device name specifies the partition to be used. Thus `"/dev/hd0/1"` specifies partition 1, `"/dev/hd0/2"` partition 2, `"/dev/hd0/3"` partition 3 and `"/dev/hd0/4"` partition 4. The special name `"/dev/hd0/0"` specifies the entire disk without honoring the partition table. This is only really useful for accessing the master boot record in sector zero, to change the partitioning. Any other kind of access may corrupt the disk.

Application code may also install a disk event callback function. It does this using the `CYG_IO_SET_CONFIG_DISK_EVENT` configuration option using the `cyg_io_set_config()`. The buffer should point to an instance of the following structure:

```
typedef void disk_channel_event_t( cyg_uint32 event, cyg_uint32 devno, CYG_ADDRWORD data );

struct cyg_disk_event_t
{
    disk_channel_event_t *event;          // Event callback function
    CYG_ADDRWORD event_data;             // Event callback user data
};

// Codes for disk_channel_event_t event argument.
#define CYG_DISK_EVENT_CONNECT           0x01
```

```
#define CYG_DISK_EVENT_DISCONNECT    0x02
```

The function should be installed on the base disk device driver (i.e. `"/dev/hd0"` rather than `"/dev/hd0/0"`) and will be called when certain events occur. At present these are restricted to `CONNECT` and `DISCONNECT` events when a new disk is attached to, or detached from, the disk controller. Support for this depends on the ability of the underlying hardware device driver being able to detect these events.

A successful lookup of the base disk device driver does not necessarily imply that removeable media is present at that time. Hardware drivers supporting removeable media will typically indicate a successful lookup, irrespective of the presence of a physical disk.

The reason for this is that the call to `cyg_io_set_config()` for the `CYG_IO_SET_CONFIG_DISK_EVENT` operation may quite reasonably occur when no physical media is present, so the lookup must succeed in order to obtain a valid I/O handle. That handle is, however, unusable for operations other than `CYG_IO_SET_CONFIG_DISK_EVENT`.

The context in which the callback is made depends on the implementation of the disk driver. It cannot be assumed that it will occur in a context in which it is safe to make complex calls. It should be assumed that the call is being made from DSR context, where blocking kernel calls may not be made. In general the function should record the details of the call and pass off control to a worker thread. See the automounter in the `FILEIO` package for an example of how this can be done.

When hardware drivers notice the `CYG_IO_SET_CONFIG_DISK_EVENT` call, they should immediately check for the presence of a disk, in case one is already connected at the time of event registration. The event callback should be made for any connected disk devices (and not for unconnected disks).

Chapter 21. Hardware Driver Interface

While the DISK I/O package provides the top level, hardware independent, part of each disk driver, the actual hardware interface is handled by a hardware dependent interface module. To add support for a new disk device, the user should be able to use the existing hardware independent portion and just add their own interface driver which handles the details of the actual device. The user should have no need to change the hardware independent portion.

The interfaces used by the disk driver and disk implementation modules are contained in the file `<cyg/io/disk.h>`.

Note: In the sections below we use the notation `<<xx>>` to mean a module specific value, referred to as “xx” below.

DevTab Entry

The interface module contains the devtab entry (or entries if a single module supports more than one interface). This entry should have the form:

```
BLOCK_DEVTAB_ENTRY(<<module_name>>,
                    <<device_name>>,
                    0,
                    &cyg_io_disk_devio,
                    <<module_init>>,
                    <<module_lookup>>,
                    &<<disk_channel>>
                    );
```

Arguments

module_name

The "C" label for this devtab entry

device_name

The "C" string for the device. E.g. `/dev/serial0`.

cyg_io_disk_devio

The table of I/O functions. This set is defined in the hardware independent disk driver and should be used exactly as shown here.

module_init

The module initialization function.

module_lookup

The device lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the controller, etc.

disk_channel

This table (defined below) contains the interface between the interface module and the disk driver proper.

Disk Controller Structure

The arrangement of disk hardware usually has a number of physical disks connected to a common controller. For example, each IDE interface connects to just two disk devices, a SCSI controller may be connected to several disks. The important feature to consider here is that any current data transfer for any one disk on a controller prevents transfers being started on any other disks on that controller until it is finished. Disk controllers are therefore the level at which concurrency and interrupt controls must be implemented.

Each disk controller is created by the macro:

```
DISK_CONTROLLER(l, dev_priv)
```

Arguments

l

The "C" label for this structure.

dev_priv

A placeholder for any device specific data for this controller.

Disk Channel Structure

Each physical disk connected to a controller is represented by a disk channel. Each channel is defined with the following macro:

```
DISK_CHANNEL(l, funs, dev_priv, controller, mbr_supp, max_part_num )
```

Arguments

l

The "C" label for this structure.

funs

The set of interface functions (see below).

dev_priv

A placeholder for any device specific data for this channel.

controller

Pointer to controller to which this disk channel is attached.

mbr_supp

Does this disk support partitioning.

max_part_num

The maximum number of partitions to be supported.

The interface from the hardware independent driver into the hardware interface module is contained in the *funcs* table. This is defined by the [DISK_FUNS macro](#).

If the space for the channel has been allocated elsewhere, the following macro may be used to initialise it:

```
DISK_CHANNEL_INIT(dc, funcs, dev_priv, controller, disk_info, part_dev_tab, part_chan_tab, part_tab)
```

The arguments are as for `DISK_CHANNEL()` except for the following:

Arguments

dc

The name of an object of type `disk_channel`. This object will be initialised by the macro.

disk_info

The name of an object of type `disk_info`.

part_dev_tab

The name of an array of objects of type `struct cyg_devtab_entry`. The number of array members must equal `max_part_num`, plus one.

part_chan_tab

The name of an array of objects of type `disk_channel`. The number of array members must equal `max_part_num`.

part_tab

The name of an array of objects of type `cyg_disk_partition_t`. The number of array members must equal `max_part_num`.

Disk Functions Structure

```
DISK_FUNS(l, read, write, get_config, set_config)
```

Arguments

l

The "C" label for this structure.

read

```
Cyg_ErrNo (*read)(disk_channel *priv, void *buf, cyg_uint32 len, cyg_uint32
block_num)
```

This function reads *len* sectors of data from the disk at the sector number given by *block_num*. The actual quantity of data transferred depends on the disk's sector size, which can be obtained using the `CYG_IO_GET_CONFIG_DISK_INFO` key.

If the read completes immediately, or the low level driver is configured to do all IO synchronously, this function will return `ENOERR`, and if it fails will return a negative error code, for example `-EIO`. If the function returns `-EWOULDBLOCK` then it has only started the transfer and will indicate its completion by calling the `transfer_done` callback.

write

```
Cyg_ErrNo (*write)(disk_channel *priv, void *buf, cyg_uint32 len, cyg_uint32
block_num)
```

This function writes *len* sectors of data to the disk at the block given by *block_num*. The actual quantity of data transferred depends on the disk's sector size, which can be obtained using the `CYG_IO_GET_CONFIG_DISK_INFO` key.

If the write completes immediately, or the low level driver is configured to do all IO synchronously, this function will return `ENOERR`, and if it fails will return a negative error code, for example `-EIO`. If the function returns `-EWOULDBLOCK` then it has only started the transfer and will indicate its completion by calling the `transfer_done` callback.

get_config

```
bool (*get_config)(serial_channel *priv, cyg_uint32 key, const void *xbuf,
cyg_uint32 *len); )
```

This function is used to get configuration data from the device. The *key* argument defines the configuration data to be fetched. The *xbuf* and **len* arguments describe a buffer into which the data will be put. The function should return `true` if the key type is supported and the buffer of sufficient length to contain the data. The value of **len* should be updated to actual length of the data returned. The function should return `false` if the driver cannot support the key value or the buffer is of insufficient length.

The following keys may be used to get information from a disk device.

CYG_IO_GET_CONFIG_DISK_INFO

This key causes a `cyg_disk_info_t` structure, as defined in `diskio.h` to be returned.

CYG_IO_GET_CONFIG_DISK_EVENT

This key returns a copy of the `cyg_disk_event_t` previously set by `CYG_IO_SET_CONFIG_DISK_EVENT`.

set_config

```
bool (*set_config)(serial_channel *priv, cyg_uint32 key, const void *xbuf,
cyg_uint32 *len);
```

This function is used to change the configuration of the device. The *key* argument defines the kind of configuration data to be set. The *xbuf* and **len* arguments describe a buffer in which the data is supplied. The function should return `true` if the key type is supported and the buffer of the correct length and the data appears valid. The function should return `false` if the driver cannot support the key value or the buffer is the wrong length, or the data is invalid in some other way.

The following keys can be sent to a driver:

CYG_IO_SET_CONFIG_DISK_MOUNT

This is invoked from the filesystem after locating the device driver to record that the device has been mounted. The generic device layer records the mount against both the partition and physical disk and passes the call on down to the driver. The *xbuf* and **len* arguments are unused.

CYG_IO_SET_CONFIG_DISK_UMOUNT

This is invoked from the filesystem to record that the device has been unmounted. The generic device layer records the unmount against both the partition and physical disk and passes the call on down to the driver. If the `chan->info->mounts` counter is zero, the driver should call the `disk_disconnected()` callback to prepare the generic layer for a potential media change. The *xbuf* and **len* arguments are unused.

CYG_IO_SET_CONFIG_DISK_EVENT

This may be invoked by the application to set a disk event callback function. The generic disk layer is mostly responsible for handling this by recording the event function in the disk channel structure. The call is additionally passed down to the hardware driver so that it may prepare the hardware, if necessary. The *xbuf* should point to a `cyg_disk_event_t` structure.

Callbacks

The interface from the hardware specific driver to the hardware independent driver is contained in a `disk_callbacks_t` structure. A pointer to this is automatically included into the disk channel structure *callbacks* field by the `DISK_CHANNEL()` macro. The `disk_callbacks_t` structure contains the following function pointers:

disk_init

```
cyg_bool (*disk_init)(struct cyg_devtab_entry *tab);
```

Initialize the disk. This must be called from the disk driver's init routine to initialize the device independent driver's data structures for this disk.

disk_connected

```
Cyg_ErrNo (*disk_connected)(struct cyg_devtab_entry *tab, cyg_disk_identify_t
*ident);
```

This is called when a valid disk device has been recognised on the given disk channel. At this point, if the disk supports partitioning the disk's partition table will be read and the partitions determined. This may be called either from the driver's init routine, for fixed disks, or alternatively from the driver's lookup routine. It may also be called from other places when, for example, disk insertion is detected. All the fields of the `ident` structure must be filled in by the driver before this call is made.

disk_disconnected

```
Cyg_ErrNo (*disk_disconnected)(struct disk_channel *chan);
```

This is called when, for example, disk removal is detected. It invalidates all the existing partition and driver information and renders the channel ready for a new disk device to be inserted.

disk_lookup

```
Cyg_ErrNo (*disk_lookup)(struct cyg_devtab_entry **tab, struct cyg_devtab_entry
*sub_tab, const char *name);
```

This must be called from the driver's lookup function to complete the lookup process. It is here that the interpretation of the partition number element of the device name is done and a new devtab entry created for the partition if necessary.

disk_transfer_done

```
void (*disk_transfer_done)(struct disk_channel *chan, Cyg_ErrNo res);
```

When the call to the `read()` or `write()` disk function returns `-EWOULDBLOCK` then the driver must indicate completion of the actual transfer by calling this function. This function should not be called from an ISR, but it may be called from the DSR.

In addition to these functions in `disk_callbacks_t`, the hardware driver is also responsible for calling the disk event callback. The calls should be made as follows:

```
disk_channel *chan = <get pointer to disk channel>;

...
```

```
chan->event( CYG_DISK_EVENT_CONNECT, devno, chan->event_data );
```

The first argument should be the event being notified: `CYG_DISK_EVENT_CONNECT` as shown here, or `CYG_DISK_EVENT_DISCONNECT`. The second argument is a device number; this is needed for devices that dynamically instantiate disk devices, such as USB. If the driver does not do this, then this argument should be `-1`. The third argument is the user data value passed in when the callback was registered.

The driver may call this function at any time and from any context other than an ISR. Normally it will be called either from a DSR or from a thread context. By default, the generic disk layer will install a dummy function in the disk channel structure, so the driver can always make the call without needing to test for a NULL pointer. A `CONNECT` event call should be made when the driver detects that a new device has been inserted into the drive, and a `DISCONNECT` event call should be made when the device is removed.

A `CONNECT` event call should also be made if a disk device is already connected when the driver observes the application registering for notification of disk events by use of the `CYG_IO_SET_CONFIG_DISK_EVENT` `cyg_io_set_config()` operation. However, this only applies to connected disks - the driver does not indicate `DISCONNECT` events for unconnected disks.

Putting It All Together

The above descriptions, while strictly useful as documentation, do not really show how it all gets put together to make a device driver. The following example of how to create the data structures for a device driver, for a standard PC target, are derived from the eCosPro IDE disk driver.

The first thing to do is to define the disk controllers:

```
static ide_controller_info_t ide_controller_info_0 = {
    ctlr:      0,
    vector:    HAL_IDE_INTERRUPT_PRI
};

DISK_CONTROLLER( ide_disk_controller_0, ide_controller_info_0 );

static ide_controller_info_t ide_controller_info_1 = {
    ctlr:      1,
    vector:    HAL_IDE_INTERRUPT_SEC
};

DISK_CONTROLLER( ide_disk_controller_1, ide_controller_info_1 );
```

A typical PC target has two IDE controllers, so we define two controllers. The `ide_controller_info_t` structure is defined by the driver and contains information needed to access the controller. In this case this is the controller number, zero or one, and the interrupt vector it uses. The `DISK_CONTROLLER()` macro generates a system defined controller structure and populates it with a pointer to the matching controller info structure.

The next step is to define the disk functions that will be called to perform data transfers on this driver. These functions the main part of the driver, together with the init and lookup functions and any ISR and DSR functions.

```
DISK_FUNS(ide_disk_funs,
          ide_disk_read,
          ide_disk_write,
```

```

        ide_disk_get_config,
        ide_disk_set_config
    );

```

We can now start generating per-disk-channel data structures. To make this easier we define a macro, `IDE_DISK_INSTANCE()` to make this easier.

```

#define IDE_DISK_INSTANCE(_number_, _ctrlr_, _dev_, _mbr_supp_) \
static ide_disk_info_t ide_disk_info##_number_ = { \
    num:        _number_, \
    ctrlr:      &ide_controller_info##_ctrlr_, \
    dev:        _dev_, \
}; \
DISK_CHANNEL(ide_disk_channel##_number_, \
    ide_disk_funs, \
    ide_disk_info##_number_, \
    ide_disk_controller##_ctrlr_, \
    _mbr_supp_, \
    4 \
); \
BLOCK_DEVTAB_ENTRY(ide_disk_io##_number_, \
    CYGDAT_IO_DISK_IDE_DISK##_number_##_NAME, \
    0, \
    &cyg_io_disk_devio, \
    ide_disk_init, \
    ide_disk_lookup, \
    &ide_disk_channel##_number_ \
); \

```

The first thing this macro does is generate an instance of the `ide_disk_info_t`. This is a driver-defined structure to contain any info that does not fit in the system defined structures. In this case the important things are the number of the device on the controller, zero or one mapping to master or slave, and a pointer to the driver-defined controller structure. The `DISK_CHANNEL()` macro creates a disk channel object and populates it with the function list defined earlier, a pointer to the matching local info structure just defined, and a pointer to the controller it is attached to. Finally, a device table entry is created. This uses linker features to install an entry into the device table that allows the IO subsystem to locate this device.

Finally we need to instantiate all the channels that this driver will support.

```

IDE_DISK_INSTANCE(0, 0, 0, true);
IDE_DISK_INSTANCE(1, 0, 1, true);
IDE_DISK_INSTANCE(2, 1, 0, true);
IDE_DISK_INSTANCE(3, 1, 1, true);

```

Each invocation of `IDE_DISK_INSTANCE()` generates all the data structures needed to access each possible physical disk that may be present.

XLI. iPAQ Framebuffer Device Driver

iPAQ Framebuffer Device Driver

Name

CYGPKG_DEVS_FRAMEBUFFER_ARM_IPAQ — eCos Support for the iPAQ framebuffer

Description

CYGPKG_DEVS_FRAMEBUFFER_ARM_IPAQ provides an eCos framebuffer device driver for the LCD panel on an iPAQ. The driver is a hardware package and is loaded automatically when configuring eCos for an iPAQ target. By default it is inactive and does not add any code size or data overheads. To activate the driver the generic framebuffer package CYGPKG_IO_FRAMEBUFFER should be added to the configuration. The driver's functionality is only accessible via the API defined by the generic package.

The driver supports a single framebuffer device, driving the LCD panel in 565 true colour mode with a resolution of 320x240 pixels at 16bpp. The `cyg_fb` structure for this is `cyg_ipaq_fb_320x240x16`, and the identifier for use with the framebuffer macro API is 320x240x16. The `ioctl` operations supported are `CYG_FB_IOCTL_BLANK_GET`, `CYG_FB_IOCTL_BLANK_SET`, `CYG_FB_IOCTL_BACKLIGHT_GET` and `CYG_FB_IOCTL_BACKLIGHT_SET`. The backlight supports 32 levels of intensity.

XLII. CSB337/900 Framebuffer Device Driver

CSB337/900 Framebuffer Device Driver

Name

CYGPKG_DEVS_FRAMEBUFFER_ARM_CSB337900 — eCos framebuffer support for a CSB337/900

Description

This package provides an eCos framebuffer device driver for a Cogent CSB337 board with a CSB900 add-on to provide the LCD panel. It has dependencies on both pieces of hardware so cannot be used with any other combination. The driver is a hardware package and is loaded automatically when configuring eCos for a csb337900 target, but not when configuring for a vanilla csb337 target. By default it is inactive and does not add any code size or data overheads. To activate the driver the generic framebuffer package CYGPKG_IO_FRAMEBUFFER should be added to the configuration. The driver's functionality is only accessible via the API defined by the generic package.

There are a number of design issues with this hardware combination. The framebuffer memory cannot be accessed in the conventional linear way, Instead the driver contains custom drawing code which will be significantly slower than the equivalent linear framebuffer routines. There are also problems with the colour handling: only three bits of control are available for each of red, green and blue intensity,

The driver supports four cyg_fb structures: cyg_csb337900_fb_240x320x8, cyg_csb337900_fb_320x240x8r90, cyg_csb337900_fb_240x320x8r180, and cyg_csb337900_fb_320x240x8r270. These all run the hardware in the same resolution but in the four different orientations, using hardware to perform the rotation. The corresponding identifiers for the macro API are 240x320x8, 320x240x8r90, 240x320x8r180 and 320x240x8r270. Obviously only one of these framebuffer devices can be used at a time. All the devices implement 8bpp non-linear displays with a writeable palette. The supported ioctl operations are CYG_FB_IOCTL_BLANK_GET, CYG_FB_IOCTL_BLANK_SET, CYG_FB_IOCTL_BACKLIGHT_GET and CYG_FB_IOCTL_BACKLIGHT_SET. The backlight can only be switched on or off, there is no support for different levels of intensity.

XLIII. Dallas DS1302 Wallclock Device Driver

Dallas DS1302 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1302 — eCos Support for the Dallas DS1302 Real-Time Clock

Description

This package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1302` provides a device driver for the wallclock device in the Dallas DS1302 Real-Time Clock chips. This combines a real-time clock and 31 bytes of battery-backed RAM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support `CYGPKG_IO_WALLCLOCK`. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as `time` and `gmtime`. The eCos C library also provides a non-standard function `cyg_libc_time_settime` for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The package provides a number of additional functions that are specific to a DS1302:

```
#include <cyg/io/wallclock/ds1302.h>

externC unsigned char cyg_wallclock_ds1302_read_tcs_ds_rs(void);
externC void cyg_wallclock_ds1302_write_tcs_ds_rs(unsigned char val);
externC void cyg_wallclock_ds1302_read_ram(int offset,
                                           unsigned char* buf, int len);
externC void cyg_wallclock_ds1302_write_ram(int offset,
                                           unsigned char* buf, int len);
```

The `_tcs_ds_rs` functions allow applications to read and update the trickle charge register in the DS1302. The manufacturer's data sheet should be consulted for further information on this register.

The `_ram` functions allow applications to read and modify the contents of the 31 bytes of battery-backed RAM. The offset specifies the starting address within the RAM and should be between 0 and 31. The buffer provides the destination or source of the data, and the length gives the number of bytes transferred. Wrap-around is not supported so the sum of the offset and length should also be less than 31. The package's `ds1302.c` testcase provides example code.

The wallclock package is initialized by a static constructor with a priority immediately after `CYG_INIT_DEV_WALLCLOCK`. Applications should not call any wallclock-related functions nor any of the DS1302-specific functions before that constructor has run.

Porting

The DS1302 is accessed via a 3-wire bus. At the time of writing there is no generic 3-wire support package within eCos, so instead the wallclock driver expects to bit-bang some GPIO lines. Typically the platform HAL provides appropriate hardware-specific macros for this, via the header file `cyg/hal/plf_io.h`. The required macros are:

```
HAL_DS1302_CE(_setting);
HAL_DS1302_SCLK(_setting);
HAL_DS1302_OUT(_setting);
HAL_DS1302_IN(_setting);
HAL_DS1302_SELECT_OUT(_setting);
```

The argument to the first three macros will always be 0 or 1 and corresponds to the desired state of the chip-enable, clock, or I/O line. For example, at the start of a transfer the wallclock driver will invoke:

```
...
HAL_DS1302_CE(1);
...
```

Asserting the CE line should activate the DS1302 chip. The `HAL_DS1302_IN` macro is used to sample the state of the I/O line and should set its argument to 0 or 1. The `HAL_DS1302_SELECT_OUT` macro is used to switch the I/O line between output (1) or input (0).

Platform HALs may provide two additional macros:

```
HAL_DS1302_DATA
HAL_DS1302_INIT();
```

`HAL_DS1302_DATA` can be used to define one or more static variables needed by the other macros, for example to hold a shadow copy of the GPIO output register. If defined, `HAL_DS1302_INIT` will be invoked during driver initialization and typically sets up the GPIO lines such that the CE and SCLK lines are outputs.

In addition the DS1302 device driver package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1302` should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

XLIV. Dallas DS1306 Wallclock Device Driver

Dallas DS1306 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306 — eCos Support for the Dallas DS1306 Real-Time Clock

Description

This package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306` provides a device driver for the wallclock device in the Dallas DS1306 Real-Time Clock chips. This combines a real-time clock and 96 bytes of battery-backed RAM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support `CYGPKG_IO_WALLCLOCK`. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as `time` and `gmtime`. The eCos C library also provides a non-standard function `cyg_libc_time_settime` for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The package provides a number of additional functions that are specific to a DS1306:

```
#include <cyg/io/wallclock/ds1306.h>

externC void cyg_wallclock_ds1306_read_regs(int offset,
                                             unsigned char* buf, int len);
externC void cyg_wallclock_ds1306_write_regs(int offset,
                                              const unsigned char* buf, int len);
externC void cyg_wallclock_ds1306_read_ram(int offset,
                                             unsigned char* buf, int len);
externC void cyg_wallclock_ds1306_write_ram(int offset,
                                             const unsigned char* buf, int len);
```

The `read_regs` and `write_regs` functions allow direct access to all of the wallclock-related registers including the alarms, the control register 0x0F, the status register 0x10, and the trickle charger register 0x11. The offset should be between 0x00 and 0x1F, specifying the first register that should be read or written. For full details of the DS1306 registers see the manufacturer's data sheet.

The `_ram` functions allow applications to read and modify the contents of the 96 bytes of battery-backed RAM. The offset specifies the starting address within the RAM and should be between 0x00 and 0x5F. The buffer provides the

destination or source of the data, and the length gives the number of bytes transferred. The package's `ds1306.c` testcase provides example code.

The wallclock package is initialized by a static constructor with a priority immediately after `CYG_INIT_DEV_WALLCLOCK`. Applications should not call any wallclock-related functions nor any of the DS1306-specific functions before that constructor has run.

Porting

The DS1306 can be either attached to an SPI bus or it can be accessed via a 3-wire interface. The driver supports both modes of operation, with a bit of support from the platform HAL. For SPI, the platform HAL should implement the CDL interface `CYGHWR_WALLCLOCK_DALLAS_DS1306_SPI` and provided an SPI device instance `cyg_spi_wallclock_ds1306`. The exact details of this device instantiation will depend on the SPI bus driver. For 3-wire the platform HAL should implement the CDL interface `CYGHWR_WALLCLOCK_DALLAS_DS1306_3WIRE` and provide a bit-bang function:

```
#include <cyg/io/wallclock/ds1306.h>

cyg_bool
hal_ds1306_bitbang(cyg_ds1306_bitbang_op op)
{
    cyg_bool result = 0;

    switch(op) {
        case CYG_DS1306_BITBANG_INIT: ...
        case CYG_DS1306_BITBANG_CE_HIGH: ...
        case CYG_DS1306_BITBANG_CE_LOW: ...
        case CYG_DS1306_BITBANG_SCLK_HIGH: ...
        case CYG_DS1306_BITBANG_SCLK_LOW: ...
        case CYG_DS1306_BITBANG_DATA_HIGH: ...
        case CYG_DS1306_BITBANG_DATA_LOW: ...
        case CYG_DS1306_BITBANG_DATA_READ: ...
        case CYG_DS1306_BITBANG_INPUT: ...
        case CYG_DS1306_BITBANG_OUTPUT: ...
    }
    return result;
}
```

The INIT operation should set the 3-wire bus to its default settings: all lines should be output low. The HIGH and LOW operations should set the specified line to the appropriate level. INPUT switches the data line from an output to an input, and OUTPUT switches it back to an output. READ should return the current state of the data line, and is the only operation for which the return value matters.

In addition the DS1306 device driver package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1306` should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

XLV. Dallas DS1390 Wallclock Device Driver

Dallas DS1390 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1390 — eCos Support for the Dallas DS1390 Serial Real-Time Clock

Description

This package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1390` provides a device driver for the wallclock device in the Dallas DS1390 Serial Real-Time Clock chips. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support `CYGPKG_IO_WALLCLOCK`. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as `time` and `gmtime`. The eCos C library also provides a non-standard function `cyg_libc_time_settime` for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

Porting

The DS1390 driver uses the SPI driver API defined by the package `CYGPKG_IO_SPI`. A suitable SPI device driver must be available for the target. The platform HAL must provide a `cyg_spi_device` structure `cyg_spi_wallclock_ds1390`. The platform HAL should initialize this structure and any associated SPI driver specific struture with the correct phase, polarity and chip select parameters for this device.

In addition the DS1390 device driver package `CYGPKG_DEVICES_WALLCLOCK_DALLAS_DS1390` should be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

Extra API Calls

In addition to the standard wallclock API calls, this driver exports a number of additional functions to permit direct access to additional features of the device. A header, `cyg/io/wallclock/ds1390.h` is available to define this API.

```
cyg_uint8 cyg_ds1390_read_reg( int addr )
```

Read and return a single 8-bit register from the DS1390, *addr* should be in the range 0x00 to 0x0F.

```
void cyg_ds1390_write_reg( int addr, int val )
```

Write a single 8-bit register to the DS1390, *addr* should be in the range 0x00 to 0x0F and *val* in the range 0x00 to 0xFF.

```
void cyg_ds1390_set_control( cyg_uint8 val )
```

Write the DS1390 control register with the content of *val*.

```
cyg_uint8 cyg_ds1390_get_control( void )
```

Read and return the value of the DS1390 control register.

```
void cyg_ds1390_set_status( cyg_uint8 val )
```

Write the DS1390 status register with the content of *val*.

```
cyg_uint8 cyg_ds1390_get_status( void )
```

Read and return the value of the DS1390 control register.

```
void cyg_ds1390_set_charger( cyg_uint8 val )
```

Write the DS1390 trickle-charge register with the content of *val*.

```
cyg_uint8 cyg_ds1390_get_charger( void )
```

Read and return the value of the DS1390 trickle-charge register.

```
int cyg_wallclock_set_alarm( cyg_uint32 secs )
```

Set the DS1390 alarm to trigger when the wallclock time matches the value of *secs*. The DS1390 alarm will match only up to days of the month, so the alarm cannot be set more than one month in the future. This function only initializes the DS1390 to generate the alarm interrupt; it is the responsibility of the caller to attach an ISR to the appropriate vector and unmask it in the interrupt controller.

XLVI. Intersil ISL12028 Wallclock Device Driver

Intersil ISL12028 Wallclock Device Driver

Name

CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028 — eCos Support for the Intersil ISL12028 Real-Time Clock

Description

This package `CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028` provides a device driver for the wallclock device in the Intersil ISL12028 Real-Time Clock chips. These combine a real-time clock, alarm functionality, and a bank of EEPROM in a single package. The driver can also be used with any other chips that provide the same interface to the clock hardware.

The package will usually be loaded into the configuration automatically whenever selecting a target which contains a compatible chip. By default it will provide the standard eCos wallclock device, although another implementation such as software emulation may be selected if desired. The only other configuration options related to this package allow users to change the compiler flags. If the application does not actually use the wallclock device, directly or indirectly, then the code should get removed automatically at link-time to ensure that the application does not suffer any unnecessary overheads.

Functionality

This wallclock device driver package implements the standard functionality required by the generic wallclock support `CYGPKG_IO_WALLCLOCK`. The functionality is not normally accessed directly. Instead it is used by the C library time package to implement standard calls such as `time` and `gmtime`. The eCos C library also provides a non-standard function `cyg_libc_time_settime` for changing the current wallclock setting. In addition RedBoot provides a **date** command which interacts with the wallclock device.

The driver does not provide direct access to any of the other functionality provided by the chip. Instead if an application wishes to access the alarms or the EEPROM memory then it can do so itself, via the generic I2C API. However any such application code does need to synchronize with the wallclock driver to prevent concurrent accesses to the device. The driver exports a mutex lock to allow for this:

```
#include <cyg/io/wallclock/isl12028.h>

extern cyg_drv_mutex_t cyg_isl12028_lock;
```

The mutex should be locked via `cyg_drv_mutex_lock` to prevent the wallclock driver from accessing the chip, and then unlocked via `cyg_drv_mutex_unlock` when the driver can safely access the chip again.

The wallclock package is initialized by a static constructor with a priority immediately after `CYG_INIT_DEV_WALLCLOCK`. Applications should not call any wallclock-related functions before that constructor has run.

Porting

The ISL12028 is accessed via an I2C serial bus, and the driver assumes the presence of the generic I2C support package `CYGPKG_IO_I2C` and a suitable hardware driver. In addition it requires that some other package, typically the platform HAL, exports a `cyg_i2c_device` structure `cyg_i2c_wallclock_isl12028`. The ISL12028 device driver package `CYGPKG_DEVICES_WALLCLOCK_INTERSIL_ISL12028` can then be included in the CDL target entry so that it gets loaded automatically whenever eCos is configured for that target.

XLVII. eCosPro™ Standard C++ library support package

Chapter 22. Introduction

This documentation describes the eCos support for the GNU standard C++ library v3 which is a component of the GNU Compiler Collection (GCC). This library, also known as libstdc++, has been designed to fully implement the requirements of the ISO 14822 standard C++ specification, and also provides some of the underlying support for language features such as C++ exceptions and run-time type identification (RTTI).

As with normal GNU toolchains, the standard C++ library is prebuilt alongside the toolchain. The library itself is not contained in this eCos package. Instead the purpose of this package is to provide any ancillary support for the library, provide the CDL definitions required for the correct operation of the library, provide a rigorous and broad test suite for the library, and of course provide this documentation.

Although the standard C++ library is part of the toolchain, some enhancements have been made to the GCC compiler specifically to support eCos, details of which are found in [Chapter 25](#).

Overview of features

The GNU standard C++ library implements virtually all the library requirements of the C++ standard. Details of the status of the library including known issues may be found on the GNU C++ Standard Library documentation pages (<http://gcc.gnu.org/onlinedocs/libstdc++/>).

In summary, the library provides support for standard C++ functionality such as:

- C++ exceptions
- Run-time type identification (RTTI) and type information
- Memory allocation routines: new, delete, allocators, etc.
- I/O streams, string streams and I/O manipulators
- C++ friendly numeric limits
- Strings and character traits
- Containers: queues, dequeues, lists, stacks, vectors, maps, sets and bitsets
- Iterators
- Algorithms such as sort, find, compare, count, replace, etc. that usually operate on containers and iterators
- Complex numbers
- Numeric arrays
- Numeric algorithms such as accumulate, inner product, partial sum, adjacent difference
- etc...

This eCos package for libstdc++ also provides support for thread-safe exceptions when using the eCos kernel, as well as expressing with CDL the requirements that the C++ library has on the rest of the eCos system. This is in fact an option within the package named `CYGPKG_LIBSTDCXX_LIBRARY`, which may be overridden and disabled, although this must be done at the developer's own risk.

This package also contains a large number of tests, including some rigorous tests of core functionality such as C++ exceptions (and in particular their thread-safety and correct operation in a multi-threaded environment), RTTI, and the main library features. These may be found in the `tests` subdirectory within this package. The GNU libstdc++ v3 test suite has also been imported and is found in the `tscpp` subdirectory.

The GNU libstdc++ implementation configures itself on the basis of underlying OS support. In a few areas, where underlying eCos support does not exist, the library configures itself to avoid the requirement for that support. This is normally of little consequence, for example due to libstdc++ providing an alternative implementation with a minor performance impact, or some trivial divergence from strict C++ standard semantics. In some cases the affected functionality is optional in the first place, for example for aspects of C99 standard support. There is one notable area which is affected however, which is that eCos contains very little support for wide characters (wchar). As such, libstdc++ configures itself to omit its own wide character interface that would have been implemented using the underlying OS wide character support. For example, this removes provision of the various `wstring` and `wstreams` classes and functions.

Chapter 23. Usage

The easiest way to start using eCos for C++ development is to use the special configuration template included in this release for this purpose. With command line configuration it may be used as in the following example for the Atmel EB40A target:

```
$ ecosconfig new eb40a libstdc++
```

If using the graphical tool, the template may be selected with the Build->Templates menu item.

Once the eCos libraries are configured and built, you may link your application. Be sure to append `-lstdc++` to the end of the link line. If you wish to use C++ exceptions, be sure to either remove `-fno-exceptions` from your compilation line, or append `-fexceptions` at the end of the compilation line. Similarly, to use RTTI, either remove `-fno-rtti` from your compilation line, or append `-frtti` to the end of the compilation line. Finally, despite what some targets may have used for their default compiler flags it is important that the option `-fvtable-gc` is *not* used.

Requirements

As noted earlier, this package uses CDL to set constraints on the rest of the eCos system for correct and standards compliant operation of the C++ library. By selecting with the libstdc++ configuration template, you will be able to start with a configuration with all the necessary packages included and options set.

Building C++ programs, particularly those that use templates heavily (either directly, or using the templates from libstdc++), can take a lot of memory on the build machine - figures in excess of 220Mb have been observed building the testsuite included with this package. Be sure to have sufficient RAM to prevent extended build times.

This package requires a patched version of the GNU compiler in order to correctly support thread-safe exceptions. Refer to the [tools building notes](#) below for further details.

Due partly to this specialised toolchain support it is not possible to use this package with the synthetic target, as native toolchains are built with specialised knowledge of the C++ runtime installed on the native OS. This is not solely due to the aforementioned patches, but also because of direct assumptions made as part of the GNU toolchain build procedure. As such, use under the synthetic target is unlikely ever to be possible.

It has been observed that GDB releases prior to GDB 6.1 can have difficulty debugging complex C++ applications, particularly those that extensively include template classes containing virtual functions. GDB 6.1 or above is recommended.

Issues to consider

Using C++ exceptions

There are a number of considerations when using C++ exceptions:

1. Care should be taken when compiling C++ code with `-fexceptions` (the default). GNU C++ has been designed so that exceptions do not necessarily add overhead to functions. However they may add overhead in the following circumstances. Namely, when:
 - a. Exceptions are actually used; OR
 - b.
 - i. The function (funcA) calls another function (funcB) with a non-null exception specifier or no exception specifier; AND
 - ii. that function (funcA) contains an automatic object; AND
 - iii. that object is defined to use a destructor, and would therefore need to have the destructor called if the called function (funcB) threw an exception through this stack frame.
2. Not all eCos API functions have null exception specifiers yet, which may lead to very small unnecessary overhead when used from C++ functions in certain circumstances [described earlier](#). Annotating all functions with null exception specifiers throughout the entirety of eCos is a massive job beyond the scope of the work done to provide C++ support. However many of the key APIs can be updated. In the current eCos sources, some obvious APIs have been updated such as all of the kernel C API and much of the ISO C/POSIX APIs.
3. Exceptions thrown from signal handlers are not supported.
4. Exceptions are not supported in ISRs, DSRs, nor ASRs. It is neither feasible nor sensible to support exceptions in ISRs or DSRs. Support for ASRs may be added at a future date, although there are no plans at present.
5. Use of C++ support from this package can increase the thread stack requirements markedly. For example, if you use C++ exceptions, you should expect to add around 4Kbytes to your stack requirements for each thread which can throw exceptions. Developers may find it useful to enable kernel thread stack overflow checking (`CYGFUN_KERNEL_THREADS_STACK_CHECKING` and possibly also `CYGFUN_KERNEL_ALL_THREADS_STACK_CHECKING`), especially if erratic behaviour is observed in threads using C++ features.

Application size

It is widely acknowledged that when using C++ libraries, memory can quickly be consumed, particularly code (ROM/Flash) space. Small targets may have difficulty running even short C++ programs. For example, it is recommended to use the MEC01 memory extension card on the Atmel AT91 evaluation board platforms (EB40, EB40A, EB42, EB55) to provide extra space for applications. The eCosPro AT91EB40A port can take advantage of the MEC01 if eCos is configured with `CYGHWR_HAL_ARM_AT91_EB40A_MEC01_RAM` enabled.

Similarly, because of the generally larger code size, download times can be lengthy if using a slow transfer mechanism such as 38400 bps serial. Alternative download options such as ethernet or fast JTAG emulator should be considered for an efficient development/debug cycle.

It may be possible in future to reduce the code size overhead using linker garbage collection more fully. Currently linker garbage collection is not performed on the GNU standard C++ library itself. However it is anticipated the savings will turn out to be small.

C++ exceptions in callbacks

eCos itself is not built with exception support (*-fexceptions*) for the reasons given earlier concerning the overhead of exception support. As a consequence throwing exceptions from callbacks will not work, e.g. when using `qsort()`, `bsearch()`, etc. eCos does not yet support a means of building individual files with differing flags - flags are manipulated only for complete packages or by using custom build rules which would be unacceptable due to maintenance overheads.

Licensing

The GNU standard C++ library has its own license distinct from that of eCos. As a basis it uses, like eCos, the GNU General Public License (<http://www.gnu.org/copyleft/gpl.html>). Also like eCos it includes an exception that permits the use of the library in proprietary applications. The exception is as follows:

As a special exception, you may use this file as part of a free software library without restriction. Specifically, if other files instantiate templates or use macros or inline functions from this file, or you compile this file and link it with other files to produce an executable, this file does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

This exception is very similar to the eCos GPL exception, and is compatible with it. Further information on this license can be found in the Introduction (<http://gcc.gnu.org/onlinedocs/libstdc++/manual/intro.html>) of the libstdc++ online documentation set.

Most of the test files imported from the libstdc++ testsuite are covered by the full GPL without any exception. This means that distributing binaries of the test executables themselves gives a requirement to make available the full source code of that binary under the terms of the GPL. This is not considered an onerous obligation as distributing test binaries for this testsuite publically is unlikely to be a common requirement.

Open issues

GCC 3.3.x issues

At this time there are only two significant known open issues that developers should be aware of which may impact development:

- GCC 3.3.x misoptimizes code in functions with complex number parameters. The workaround is to compile without `-O2` (or append `-O0` to the end of the compile line). This issue has been filed with the GCC project as bug #15061 (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=15061). As a consequence of this compiler bug, the `complex2` test in this package is likely to fail.
- GCC 3.3.x fails to return `NULL` when using the `std::nothrow` variant of the `new` operator when an amount of memory is requested beyond what the system has available. Instead of `NULL`, 4 is returned. This is listed with the GCC project as bug #13215 (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=13215) and the problem is not going to be addressed in the GCC 3.3.x series. The problem is fixed in GCC 3.4. As a consequence of this compiler bug, the `new1` test in this package will fail with GCC 3.3.x.

GCC 3.4.x issues

The only known issue affecting the use of GCC 3.4.x is specific to M68K/Coldfire, where the software floating point emulation is too imprecise, and causes a small number of tests within the libstdc++ package to fail, primarily those that test long double support.

Generic issues

It is worth mentioning that, as previously mentioned above, wide character support is not included. Support for wide characters may be developed in due course, but it would require significant development in the underlying eCos C library.

Chapter 24. Testing

As noted earlier, tests have been written to verify the operation of certain specific areas of interest in the C++ library support, particularly the use of exceptions from multiple threads which is addressed in the `throw*` tests.

The GNU libstdc++ v3 testsuite has been imported into this package and may be found under the `tscpp` subdirectory of this package. There are a large number of tests within the libstdc++ testsuite of varying rigour. An analysis of the coverage has been made, and any notable gaps in the test coverage have been addressed in the custom tests in the `tests` subdirectory of this package.

The testsuite is quite large and takes some time to build, and so although built by default it may disabled with the `CYGPKG_LIBSTDCXX_OFFICIAL_TESTSUITE` CDL option. Some tests contain aspects which only operate if the RAM filesystem package is enabled, therefore to test the library more thoroughly developers may wish to consider enabling the RAM filesystem.

Notes on how the libstdc++ testsuite was imported, including what types of changes were made and what the results were, are available within this package in the `tscpp/NOTES` text file.

Chapter 25. Toolchain

To build GCC for use with this package, it is necessary to follow some additional steps compared with what would ordinarily be required for building the compiler. These steps are required to provide the eCos header files which are used by the GCC build, to determine properties of the run-time system and to apply a set of changes (a “patch”) to allow eCos to provide C++ exception support in a flexible and future-proof way. This patch takes particular care to ensure that the compiler and libstdc++ continue to behave correctly when no eCos kernel is present.

1. With eCos installed, the ECOS_REPOSITORY environment variable set and **ecosconfig** in your PATH variable, run the following commands at a **bash** shell prompt in an empty directory, choosing a TARGET of the appropriate architecture:

```
$ ecosconfig new TARGET libstdc++
$ ecosconfig tree
$ make headers
```

2. Take the header tree generated under `install/include` and install it in the `TARGET/sys-include` subdirectory where you intend to install your tools. For example if you wish to install the new tools to `/opt/newtools`, then place the headers in a new directory `/opt/newtools/TARGET/sys-include`. (Note you must ensure you have write-access to `/opt` in this example, or you can choose an alternate path).
3. A C++ exception support patch is supplied on the eCosPro Developer’s Kit CD-ROM. Once the patch has been applied, it is necessary to run the following command:

```
$ contrib/gcc_update --touch
```

4. Configure GCC from within an empty build directory as follows, ensuring that the GNU binary utilities are at the head of the PATH:

```
$ /src/gcc-3.x.x/configure --target=TARGET \
    --prefix=/opt/newtools --enable-languages=c,c++ \
    --with-gnu-as --with-gnu-ld --with-newlib \
    --enable-threads
```


XLVIII. gcov Test Coverage Support

Test Coverage

Name

CYGPKG_GCOV — eCos Support for the gcov test coverage tool

Description

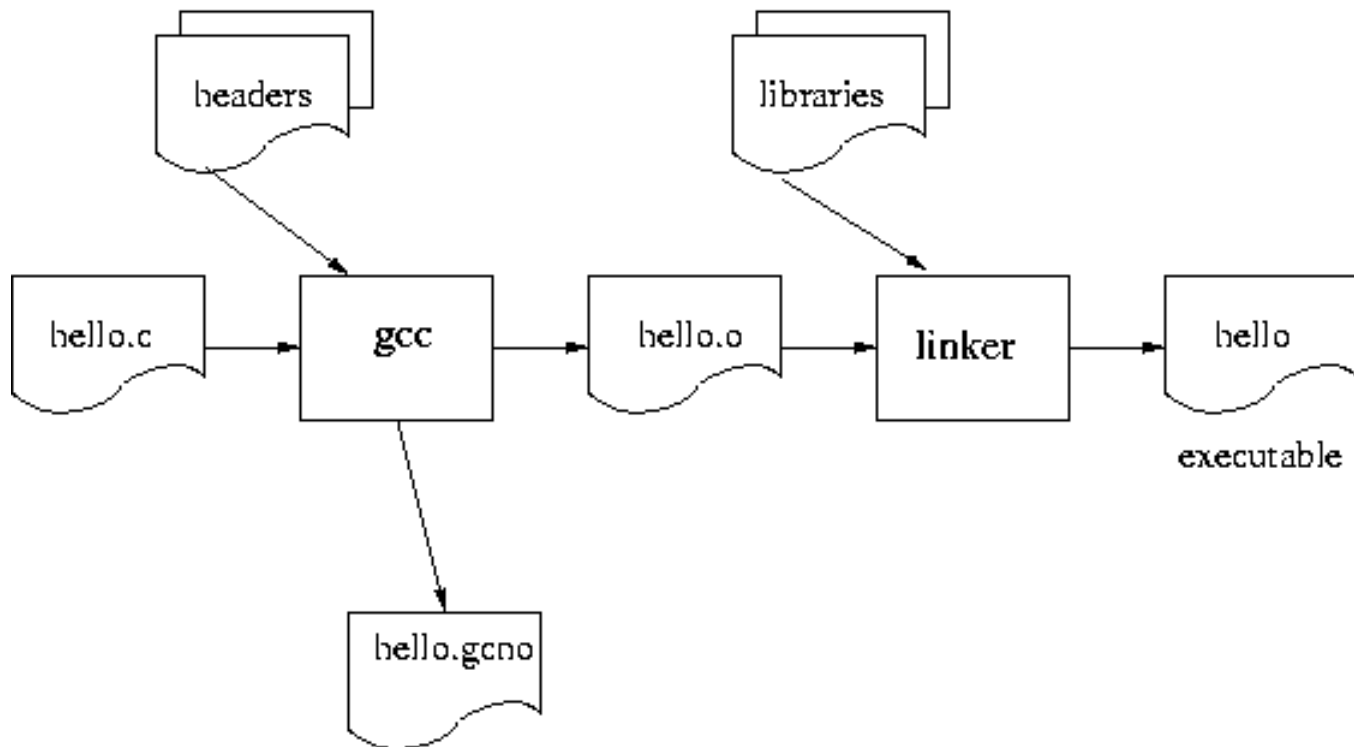
The GNU gcov tool provides test coverage support. After a test run it can be used to find code that was never actually executed. The testing conditions can then be adjusted for another test run to ensure that all the code really has been tested. The tool can also be used to find out how often each line of code was executed. That information can help application developers to determine where cpu time is being spent, and optimization effort can be focussed on critical parts of the code.

A typical fragment of gcov output looks something like this:

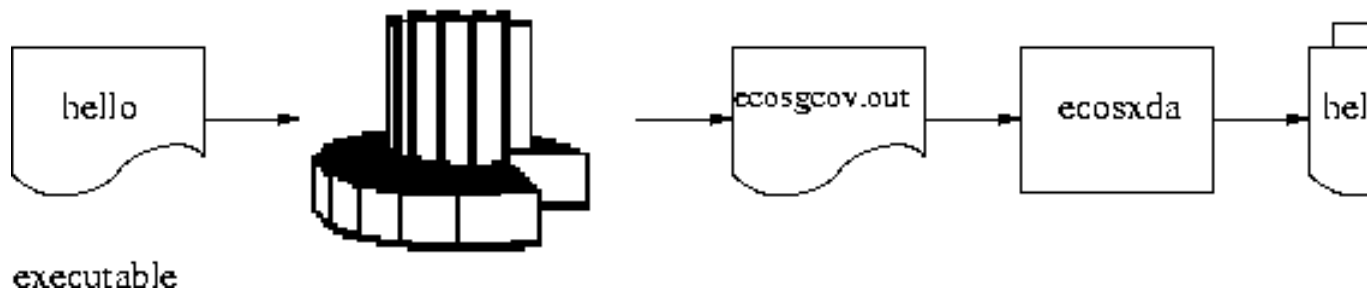
```
      80002:  60:  for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
-: 61:  {
-: 62:
      80000:  63:      Proc_5();
      80000:  64:      while (Int_1_Loc < Int_2_Loc) /* loop body executed once */
-: 65:  {
      80000:  66:          Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
      80000:  67:          Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
      80000:  68:          Int_1_Loc += 1;
-: 69:  } /* while */
      240000:  70:      for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
-: 71:          /* loop body executed twice */
-: 72:  {
      160000:  73:          if (Enum_Loc == Func_1 (Ch_Index, 'C'))
-: 74:              /* then, not executed */
-: 75:          {
      #####:  76:              Proc_6 (Ident_1, &Enum_Loc);
      #####:  77:              strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
      #####:  78:              Int_2_Loc = Run_Index;
      #####:  79:              Int_Glob = Run_Index;
-: 80:          }
-: 81:  }
-: 82:      ...
-: 83:  }
```

Each line show the execution count and line number. An execution count of -: means that there is no executable code at that line. In this example the main loop is executed 80000 times. The body of the inner `for` loop is executed more often, but the `if` condition never triggers so four lines of code have not been tested.

The gcov tool works in conjunction with the gcc compiler. Application code should be built with two additional compiler flags `-fprofile-arcs` and `-ftest-coverage`. The first option causes the compiler to generate additional code which counts the number of times each basic block is executed. The second option results in additional files with `.gcn` suffixes which allow gcov to map these basic blocks onto lines of source code. Older versions of the compiler used to generate files with `.bb` and `.bbg` suffixes instead.



The resulting executable can be run on the target hardware as usual. The basic block counting will initialize automatically and the counts will accumulate. If gcov is used for native development rather than for embedded targets then these counts will be written out to `.gcda` data files automatically when the program exits (older versions of the compiler used to generate files with `.da` suffixes). A typical embedded target will not have access to the host file system so a different approach must be used. The counts can be extracted from the target using either a gdb macro or by a tftp transfer, giving a single `ecosgcov.out` file with counts for the entire application. This file should then be processed with the `ecosxda` script to give count files for each application source file.

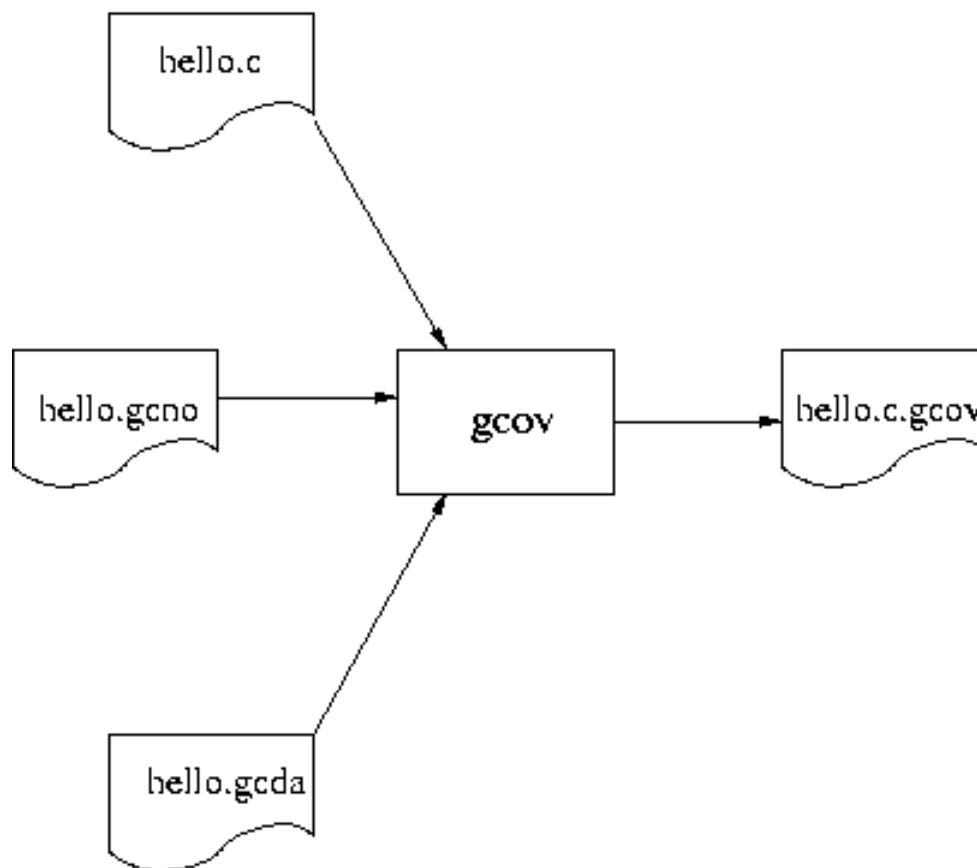


It is now possible to run gcov on each source file. The exact format of the various files varies with the compiler version so it is important to use the version of gcc that comes with the compiler.

```

$ m68k-elf-gcov dhrystone.c
 89.25% of 214 source lines executed in file dhrystone.c
Creating dhrystone.c.gcov.
#
  
```


gcov will read in the basic block counts from the generated `.gcda` file. These basic blocks are mapped onto the source code using the information in the `.gcno` files.



gcov provides various options, for example it can output summaries for each function. Full details of the available functionality can be found in the gcov section of the gcc documentation.

Building Applications for Test Coverage

To perform application test coverage the gcov package `CYGPKG_GCOV` must first be added to the eCos configuration. On the command line this can be achieved using:

```
$ ecosconfig add gcov
$ ecosconfig tree
$ make
```

Alternatively the same steps can be performed using the graphical configuration tool. The package only has two configuration options related to tftp transfers, described [below](#).

In addition application code should be compiled with two additional options, `-fprofile-arcs` and `-ftest-coverage`. The first option causes the compiler to insert additional code for basic block counting, plus an initialization call to `__gcov_init_func()` which is provided by the eCos gcov package. The second option

results in additional `.gcno` output files which `gcov` will need later. The target-side memory needed to store the basic block counts is allocated statically.

When code is compiled with optimization the compiler may rearrange some of the code, if that leads to better performance. Sometimes this causes the `gcov` output to be rather confusing. Compiling with `-O0`, thus disabling optimization, can help.

Extracting the Data

The basic block counts must be extracted from the target and saved to a file `ecosgcov.out` on the host. This package provides two ways of doing this: a `gdb` macro or `tftp` transfers. Using `tftp` is faster but requires a TCP/IP stack on the target. It also consumes some additional target-side resources, including an extra `tftp` daemon thread and its stack. The `gdb` macro can be used even when the eCos configuration does not include a TCP/IP stack. However it is much slower, typically taking several minutes to retrieve all the counts for a non-trivial application.

The `gdb` macro is called `gcov_dump`, and can be found in the file `gcov.gdb` in the `host` subdirectory of this package. A typical way of using this macro is:

```
(gdb) source <repo>/services/profile/gcov/<version>/host/gcov.gdb
(gdb) gcov_dump
```

This macro can be used any time after the application has initialized, and will store the counts accumulated so far to the file `ecosgcov.out` in the current directory. The counts are not reset.

If the configuration includes a TCP/IP stack then the data can be extracted using `tftp` instead. There are two relevant configuration options. `CYGPKG_GCOV_TFTPD` controls whether or not `tftp` is supported. It is enabled by default if the configuration includes a TCP/IP stack, but can be disabled to save target-side resources. `CYGNUM_GCOV_TFTPD_PORT` controls the UDP port which will be used. This port cannot be shared with other `tftp` daemons. If neither application code nor any other package (for example the `gprof` profiling package) provides a `tftp` service then the default port can be used. Otherwise it will be necessary to assign unique ports to each daemon.

Using `tftp` requires some additional code in the application. Specifically the daemon cannot be started until the network is up and running, and that usually happens at the behest of application code rather than automatically. The following code fragment illustrates what is required:

```
#include <pkgconf/system.h>
#include <network.h>
#ifdef CYGPKG_GCOV
# include <pkgconf/gcov.h>
# include <cyg/profile/gcov.h>
#endif
...
int
main(int argc, char** argv)
{
    ...
    init_all_network_interfaces();
#ifdef CYGPKG_GCOV_TFTPD
    gcov_start_tftpd();
#endif
}
```

```
    ...
}
```

The data can then be retrieved using a standard tftp client. There are a number of such clients available with very different interfaces, but a typical session might look something like this:

```
$ tftp
tftp> connect 10.1.1.134
tftp> binary
tftp> get ecosgcov.out
Received 138740 bytes in 1.7 seconds
tftp> quit
```

The address 10.1.1.134 should be replaced with the target's IP address.

ecosxda

gcov expects separate .gcda files for each application source file compiled with `-fprofile-arcs`. However it would be inconvenient to extract each .gcda file via tftp or a gdb macro. Instead the data is first written to a single file `ecosgcov.out`. The `ecosxda` utility script should then be used to process `ecosgcov.out` and generate the .gcda files.

The `ecosxda` script can be found in the `host` subdirectory of this package. Since it is a simple Tcl script it does not need to be built or installed. If desired it can be copied to a suitable location on the user's PATH. Alternatively the subdirectory contains suitable `configure` and `Makefile.in` files, allowing the script to be installed automatically as part of the generic eCos host-side build system. The toplevel file `README.host` contains more information about this.

Typically `ecosxda` will be invoked with no arguments.

```
$ ecosxda
```

It will read in an `ecosgcov.out` file from the current directory and output or update the .gcda files appropriate for the application. If a given .gcda file already exists then by default `ecosxda` will read it in and merge the old and new counts, rather than write a new set. This allows data from several test runs to accumulate, giving more comprehensive test coverage. Merging the counts is only possible if the source file has not been recompiled, otherwise the old counts will be discarded to avoid contaminated results.

`ecosxda` takes a number of command line options.

```
-h
```

```
--help
```

Provide brief usage information.

```
-V
```

```
--version
```

Display the version of the `ecosxda` script being used.

`-v`

`--verbose`

Provide additional diagnostic output. Repeated uses increase the level of verbosity.

`-n`

`--no-output`

Do not actually create or modify any `.gda` files. Typically this is used to find out whether any files would be replaced rather than merged, and it can also be used to validate the `ecosgcov.out` file.

`-r`

`--replace`

This forces `ecosxda` to ignore any existing counts in the `.gda` files, rather than try to merge the existing and new counts. Typically it is used to discard the results from previous test runs.

In addition it is possible to specify the file containing the new counts, instead of the default `ecosgcov.out`. This may prove useful if several sets of results are extracted to different files during a single test run, to determine what code gets run at various stages. For example:

```
$ ecosxda -r stage2.out
```

Directories and eCos Test Coverage

In a simple build environment the source code, the `.gcn` files generated by the compiler, and the `.gda` files output by `ecosxda`, will all reside in the same directory. That makes it easy for `gcov` to find the various files it needs. `gcov` will also generate its `.gcov` files in the same directory.

In more complicated build environments the source code may be kept completely separate from the build tree. eCos itself provides an example of this: the source code is held in a clean component repository, and builds happen in separate build trees. To use `gcov` in such an environment it is necessary to understand what files will be created where:

1. The compiler will output the `.gcn` file in the same directory as the object file. For an eCos build tree this will be below the version directory of each package. For example, if the kernel source file `sync/mutex.cxx` is built with `-ftest-coverage` then the `kernel/current/src/sync` subdirectory in the build tree will contain the `mutex.gcn` files.
2. The `.gda` files will end up in the same directory as the `.gcn` files. The compiler puts the full path name in each object file, and this path is copied into the `ecosgcov.out` file and used by `ecosxda`. It is assumed that `ecosgcov.out` will be processed on the same machine that was used to compile the code.
3. When `gcov` is invoked it can be given a full pathname for the source file. By default it assumes that the other files will be in the current directory, but a `-o` command line option can be used to override this.
4. `gcov` will output its `.gcov` files in the current directory.

To perform test coverage of eCos itself, in addition to or instead of the application, it is necessary to rebuild eCos with the appropriate flags. This involves changing the configuration option `CYGBLD_GLOBAL_CFLAGS` to include `-ftest-coverage` and `-fprofile-arcs`, then performing a clean and a full make. The basic block counts can

be extracted and processed with `ecosxda` as before, and the `.gcno` and `.gcda` files will all end up in the build tree. This test coverage data can then be processed in the build tree using, for example:

```
$ cd <build>
$ cd kernel/<version>
$ m68k-elf-gcov -o . <repo>/kernel/<version>/sync/mutex.cxx
...
 58.50% of 147 source lines executed in file <repo>/kernel/current/src/sync/mutex.cxx
Creating mutex.cxx.gcov.
```

Where `<build>` is the location of the build tree and `<repo>` is the location of the eCos component repository.

Additional Target-side Functions

The eCos gcov package provides a small number of additional target-side functions. Prototypes for these are provided in the header file `<cyg/profile/gcov.h>`.

```
...
int
main(int argc, char** argv)
{
    ...
    init_all_network_interfaces();
#ifdef CYGPKG_GCOV_TFTPD
    gcov_start_tftpd();
#endif
    ...
}
```

If the eCos configuration includes a TCP/IP stack and if a target-side tftp daemon will be used to extract the data from the target to the host then application code should call `gcov_start_tftpd` once the network is up. This cannot be done automatically by the gcov package itself since that package has no simple of detecting when the network is ready.

```
extern void gcov_reset(void);
```

This function can be used to reset all basic block counts. If the application operates in a number of distinct stages then it may be useful to get coverage data for each stage, rather than a single set of results for the whole test run. It can also be used to get test coverage for a specific sequence of external inputs.

64-bit arithmetic is used for the basic block counts. Hence it should not be necessary to perform occasional resets to avoid counters overflowing.

To operate properly `gcov_reset` needs to disable interrupts for a while, so it should not be used in situations which require hard real-time performance.

```
extern void gcov_dump(void);
```

Test Coverage

This is a utility routine which outputs some of the basic block information via `diag_printf` calls. It is intended primarily to help with debugging the gcov code itself.

In addition the `<cyg/profile/gcov.h>` header exports the data type `gcov_module` and a variable `gcov_head` which acts as the head of a linked list of `gcov_module` structures. This allows application code to access and manipulate the basic block data directly, if desired.

XLIX. Robust Boot Loader

Robust Boot Loader

Name

CYGPKG_RBL — provide a robust boot service

Description

The Robust Boot Loader (RBL) package provides an alternative to the usual RedBoot facilities for managing flash hardware, the **fis** and **fconfig** commands. It provides the following facilities:

1. An application can be stored in flash memory. This can be loaded and run automatically during RedBoot startup, or it can started manually.
2. The application can be updated in a robust fashion. The RBL code will automatically maintain a backup copy of the previous version of the application in flash. If something goes wrong during the update, for example a power cut, the system can still boot up using the backup.
3. There is also support for a block of persistent data. Applications can save a block of memory to flash and restore it later, possibly after a reboot. Again updates will happen in a robust fashion with a primary and a backup copy.
4. The RBL functionality can be accessed by application code via a suitable [API](#). Some of the functionality is also available via [RedBoot commands](#).

The RBL package is aimed primarily at field deployment of production systems rather than at application development. The code update facility allows the application to be updated when necessary. The persistent data can be used to hold per-unit settings and any state that should be preserved across power failures or anything else that causes a reboot. There is also some control over standard I/O: in a deployed system the serial port may be connected to some other hardware and outputting a RedBoot banner during startup could be confusing.

The package comes in two parts. When building RedBoot for a particular target platform the package detects the presence of `CYGPKG_REDBOOT` in the configuration and modifies the available functionality. In any other configuration, used for building the application, the package provides a set of library routines that allow access to this functionality.

The RBL code has gone through a number of versions, affecting the protocol used between the RedBoot code and the application. The version is selected via the configuration option `CYGNUM_RBL_VERSION`. RedBoot and the application must use the same version of RBL: an application linked against a V1 version of the RBL library routines will not work on top of a RedBoot built with the V2 code, and vice versa. By default the latest version of the protocol will be used.

RedBoot Builds

The first step in building RedBoot with RBL support is to create a RedBoot configuration appropriate for the target platform. This is somewhat target specific so the appropriate platform HAL or RedBoot documentation should be consulted for further details. Typically RedBoot should be configured for ROM startup.

Given this initial configuration the RBL package `CYGPKG_RBL` should now be added to the configuration, using one of the eCos configuration tools. For example with the command line tool this involves using `ecosconfig add rbl`.

This involves a conflict related to the configuration option `CYGPKG_REDBOOT_FLASH`. The RBL code replaces the standard RedBoot flash support, based around the **fis** and **fconfig** commands, so that must be disabled.

There are a number of other configuration options which may need to be changed at this point:

`CYGNUM_RBL_VERSION`

This determines the protocol used between the RedBoot code and the application library routines. The default is to use the latest version. If RedBoot should continue to work with existing application binaries using an older version then this configuration option should be updated to match.

`CYGNUM_RBL_FLASH_BASE`

This option is only of interest if the target hardware has multiple banks of flash. The current version of the package requires that all RBL code and data blocks reside in a single bank. By default this will be the same bank of flash that holds RedBoot, either as determined by `CYGNUM_REDBOOT_FLASH_BASE` or the first bank of flash. If `CYGNUM_RBL_FLASH_BASE` is enabled then its value will determine the bank of flash used for the RBL code and data blocks. This may be useful if for example the board has a small NOR flash for holding RedBoot and a much larger serial dataflash for holding the eCos application and its data.

`CYGDAT_RBL_RESERVED_FLASH_BLOCKS`

Some of the flash blocks should not be used by the RBL package to store code or data. On a typical system it will be necessary to reserve at least flash block 0 because that is used to hold RedBoot itself. If a single flash block is too small to hold RedBoot or if there are other blocks which should be reserved then these should be listed in the value of `CYGDAT_RBL_RESERVED_FLASH_BLOCKS`, using a comma-separated list of numbers.

`CYGNUM_RBL_CODE_BLOCKS`

`CYGNUM_RBL_DATA_BLOCKS`

The RBL code inside RedBoot needs to know how many flash blocks to allocate for the application code and for the persistent data. Because it maintains both primary and backup versions the actual requirements will be double the sum of these two options. These values will be hard-coded into the versions of RedBoot that get deployed in the field and cannot easily be changed, so they should be chosen carefully.

`CYGDAT_REDBOOT_DEFAULT_IP_ADDR`

Because the flash is used for storing RBL blocks there is nowhere for Redboot to store an **fis** directory or any **fconfig** settings. Hence certain settings like the default IP address cannot be managed via **fconfig**. Instead such settings must be configured statically via configuration options. Typically this will not be a problem because RBL will be used primarily in systems deployed in the field and RedBoot is used only to start the application. If application code needs such settings then they can be held in the RBL persistent data.

`CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT`

`CYGNUM_REDBOOT_BOOT_SCRIPT_DEFAULT_TIMEOUT`

Usually RedBoot will obtain its boot script, if any, from the **fconfig** data. This is not available when using RBL so instead the boot script should be specified using a configuration option. For a deployed system `CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT` should be set to `"rbl boot\n"`. If the flash contains a valid application then this will be loaded and run automatically.

RedBoot gives users a chance to interrupt the system before running the boot script. Typically this is irrelevant for a deployed system because there will be nothing attached to the terminal port, but it can be useful during development if for example a broken version of the application has been installed by mistake. For a deployed

system it may be desirable to reduce the timeout from the default 10 seconds, so that the application restarts more quickly after a power failure.

CYGGLO_RBL_STDIO

This provides control over standard I/O behaviour and is described in more detail [below](#).

Once RedBoot has been appropriately configured it can be built and installed as usual for the target platform.

Application Builds

In addition to the RedBoot extensions the RBL package provides a number of [functions](#) for use by application code. These functions interact with the main RBL code inside the currently installed RedBoot using the eCos virtual vector mechanism.

The RBL package is not part of any standard eCos configuration so it must be added explicitly to the configuration used for application builds, for example by using **ecosconfig add**. The package's CDL script will detect that CYGPKG_REDBOOT is not defined and hence know that the package is being used for an application build rather than for extending RedBoot. The package does not require any special support from other parts of eCos.

The package's `misc` subdirectory contains three example programs that illustrate the use of the RBL API, together with a `README` and various support files.

Standard I/O

The RBL package builds on standard RedBoot functionality such as boot scripts. Some of this functionality is desirable in a development environment but can cause problems for a system deployed in the field. For example during startup RedBoot will usually output a banner message via a serial port, and it will listen on that serial port for an incoming control-C character in case the developer wants to abort the boot script. When a system is deployed that serial port may be connected to other hardware which does not expect the banner message, or which might be sending a stream of data that happens to include the occasional control-C.

When configuring and building RedBoot it is possible to change the default standard I/O behaviour using the configuration option `CYGGLO_RBL_STDIO`. This can take one of three values:

`standard`

RedBoot I/O behaves as usual, so typically the RedBoot banner will be sent out of a serial port and RedBoot will abort the boot script if it detects an incoming control-C. Application standard I/O is also not affected so for example a `printf` call will result in data being sent out of the serial port.

`suppress_redboot`

RedBoot I/O is suppressed. Any RedBoot output such as the banner message will be discarded, and incoming characters will be ignored. When the application is started via **rbl boot** I/O is reset so the application behaves as normal.

`suppress_all`

RedBoot I/O is suppressed as before, but I/O is not reset when the application is started. Hence any `printf` or similar output produced by the application gets discarded as well (at least in the default configuration where output is sent via the HAL diagnostic pseudo-device). The serial port is now available for other purposes, for example it can be accessed via a full serial driver.

Suppressing application I/O in this way will only work if the application uses the HAL virtual vector mechanisms to route I/O activity via RedBoot. If instead the application is configured to ignore the virtual vectors, for example by disabling `CYGSEM_HAL_USE_ROM_MONITOR`, then the RedBoot `CYGGLO_RBL_OUTPUT` setting will have no effect on application I/O.

Manipulating the standard I/O behaviour like this should only be done when the application will be started automatically via an **rbl boot** command in the boot script. If instead applications will be run via a gdb session interacting with RedBoot then suppressing RedBoot I/O will interfere with the gdb traffic.

There is also a common HAL configuration option that application developers should be aware of: `CYGDBG_HAL_DEBUG_GDB_CTRL_C_SUPPORT`. By default this will be enabled for a RAM startup application. It causes the system startup code to install an interrupt handler that looks for incoming control-C characters and switch control to the gdb stubs. Usually this is sensible behaviour during development, but the option should be disabled for a deployed system. Note that it is the application configuration that needs to be changed, not the RedBoot configuration.

RedBoot Commands

Name

rbl — access RBL functionality via the RedBoot prompt

Synopsis

rbl info

rbl newcode -b <buffer> -l <length>

rbl newdata -b <buffer> -l <length>

rbl boot

rbl condboot

Description

A RedBoot configured with RBL support will provide a new command **rbl** with various sub-commands. These allow users to access the RBL functionality at the RedBoot prompt.

rbl info can be used to get information about the RBL subsystem, for example how many flash blocks are allocated to each code block. Typical output might look like:

```
RedBoot> rbl info
Code block A : backup
  First flash block 1 (address 0xffe40000)
  Size 52012, sequence number 3
Code block B : primary
  First flash block 3 (address 0xffec0000)
  Size 52028, sequence number 4
Data block A : primary
  First flash block 2 (address 0xffe80000)
  Size 272, sequence number 11
Data block B : backup
  First flash block 5 (address 0xffff40000)
  Size 272, sequence number 10
```

This shows that the code is currently on its fourth revision and is held in flash block 2 at the given address. Here all code and data blocks fit into a single flash block, which will not always be the case. The data is currently on revision 11.

The **rbl newcode** command can be used to install a new revision of the code, although usually this will be done by the application itself via a call to [rbl_update_code](#). However the RedBoot command can be used for the initial installation or if the currently installed version is broken somehow. Typically the code will first be loaded into RAM using a RedBoot **load** command, then programmed into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
Raw file loaded 0x0000d400-0x00019f3b, assumed entry at 0x0000d400
xyzModem - CRC mode, 409(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot> rbl newcode -b %{freememlo} -l 52028
... Erase from 0xffe40000-0xffe80000: .
... Program from 0x0000d400-0x00019f3c at 0xffe40000: .
... Program from 0x00005728-0x0000573c at 0xffe7ffec: .
```

The loaded program should be a stripped ELF executable appropriate for the target platform. The **-b** option specifies the memory location. Usually the RedBoot `%{freememlo}` variable will be used for this. The **-l** option corresponds to the file length. First the appropriate flash block or blocks are erased. Next the code is written to the flash. Finally the RBL code writes a little trailer at the end of the flash block containing a checksum, a sequence number, and similar information.

The **rbl newdata** command provides the same functionality for persistent data. Again the data is first loaded into RAM, then programmed into flash.

```
RedBoot> load -r -m ymodem -b %{freememlo}
Raw file loaded 0x0000d400-0x0000d5ba, assumed entry at 0x0000d400
xyzModem - CRC mode, 6(SOH)/0(STX)/0(CAN) packets, 3 retries
RedBoot> rbl newdata -b %{freememlo} -l 443
... Erase from 0xffff40000-0xffff80000: .
... Program from 0x0000d400-0x0000d5bb at 0xffff40000: .
... Program from 0x00005728-0x0000573c at 0xffff7ffec: .
```

The **rbl boot** command is used to load and run the current primary code block. The block should contain a stripped ELF executable which still contains the required relocation tables and the entry point, so there is no need for additional options. During development this command can be run manually to try out the current version of the application. In a production system RedBoot can be configured to run this command automatically by setting the configuration option `CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT`. The command does not return. If it is necessary to get back to a RedBoot prompt then either the target board should be reset or the loaded application should call [rbl_reset](#).

rbl condboot is a variant of **rbl boot**, available only on certain platforms. The command checks a platform-specific condition, for example the state of a jumper or a button. Depending on the condition **condboot** will either proceed to load and run the current primary code block in exactly the same way as **rbl boot**, or it will do nothing. Again in a production system RedBoot can be configured to run this command automatically by setting the configuration option `CYGDAT_REDBOOT_DEFAULT_BOOT_SCRIPT`. Following power up or reset RedBoot will normally run the current application, but if the button is held down then it will provide an interactive session instead (unless of course the boot script runs additional commands after **rbl condboot**). The interactive session allows the usual RedBoot and **rbl** commands to be executed, so for example the user can perform a ymodem transfer and then replace the primary code block via **rbl newcode**.

Jumpers and buttons are inherently platform-specific so **rbl condboot** will only be built if the platform HAL provides a suitable macro `HAL_RBL_CONDBOOT`. Typically this macro would be defined in the header file

cyg/hal/plf_io.h which is automatically #include'd by the RBL code. The macro should take the following form:

```
#define HAL_RBL_CONDBOOT(_do_boot_) \
    CYG_MACRO_START \
    ... \
    CYG_MACRO_END
```

`_do_boot_` should be set to 1 if the system should proceed with the bootstrap, i.e. load and run the primary code block. It should be set to 0 if **rbl condboot** should do nothing. Depending on the complexity of the hardware the macro body may involve just a couple of lines of inline code or it may involve a function call into the main platform HAL code, for example:

```
#define HAL_RBL_CONDBOOT(_do_boot_) \
    CYG_MACRO_START \
    extern int hal_alaia_rbl_condboot(void); \
    _do_boot_ = hal_alaia_rbl_condboot(); \
    CYG_MACRO_END
```


Application Library

Name

RBL functions — allow applications to access RBL services

Synopsis

```
#include <cyg/rbl/rbl.h>

cyg_bool rbl_get_flash_details(rbl_flash_details* details);
rbl_flash_block_purpose rbl_get_flash_block_purpose(cyg_uint32 block);
cyg_bool rbl_get_block_details(rbl_block_purpose which, rbl_block_details* details);
cyg_bool rbl_update_code(void* buffer, cyg_uint32 length);
cyg_bool rbl_update_data(void* buffer, cyg_uint32 length);
cyg_bool rbl_load_data(void* buffer, cyg_uint32 length);
void rbl_reset(void);
```

Flash Details

`rbl_get_flash_details` can be used by application code to get information about the flash hardware and how it is being used by the RBL code inside RedBoot. The function takes a single argument, a pointer to an `rbl_flash_details` structure.

```
typedef struct rbl_flash_details {
    cyg_uint8*   rbl_flash_base;
    int          rbl_flash_block_size;
    int          rbl_flash_num_blocks;
    int          rbl_code_num_blocks;
    int          rbl_data_num_blocks;
    int          rbl_trailer_size;
} rbl_flash_details;
```

The `rbl_flash_base` field gives the location of the flash in the target's memory map. Application code does not usually need this information since the flash hardware is entirely managed by RedBoot, but it may be useful for debugging purposes.

The `rbl_flash_block_size` and `rbl_flash_num_blocks` provide further information about the flash hardware. Typical sets of values might be 8 blocks of 256K apiece, or 64 blocks of 64K. If the flash chips support smaller boot blocks then the eCos flash management code will usually treat these as a single full-size block.

`rbl_code_num_blocks` gives the number of flash blocks that will be used for the primary and the backup code blocks. It corresponds to the value of the `CYGNUM_RBL_CODE_BLOCKS` configuration option used when building RedBoot. `rbl_data_num_blocks` provides the same information for the persistent data blocks, with a value of 0 indicating that the support for persistent data was disabled.

The RBL code uses a small amount of flash memory for management purposes. This amount of memory is given by `rbl_trailer_size`. Application code can determine the maximum size of an executable using:

```
rbl_flash_details details;
if (! rbl_get_flash_details(&details)) {
    fputs("Error: failed to get RBL flash details\n", stderr);
    return false;
}
size = (details.rbl_flash_block_size * details.rbl_code_num_blocks) -
    details.rbl_trailer_size;
```

The `rbl_get_flash_details` function will return true on success, false on failure. The most likely reason for failure is that the current RedBoot installation does not have RBL support.

Flash Blocks

`rbl_get_flash_block_purpose` can be used to find out how the RBL code inside RedBoot has allocated each flash block. Typically this is used only for debugging purposes. The argument should be a small number between 0 and `details.rbl_flash_num_blocks - 1`. The return value will be one of the following:

`rbl_flash_block_reserved`

This flash block is reserved, for example it may hold some or all of the RedBoot code. Flash blocks can be reserved using the configuration option `CYGDAT_RBL_RESERVED_FLASH_BLOCKS` when building RedBoot.

`rbl_flash_block_code_A`
`rbl_flash_block_code_B`
`rbl_flash_block_data_A`
`rbl_flash_block_data_B`

The flash block is used for an RBL code or data block.

`rbl_flash_block_free`

This flash block is not used by the RBL code. It may be used by application code for other purposes.

`rbl_flash_block_invalid`

This value will be returned if the argument to `rbl_get_flash_block_purpose` is outside the valid range. It will also be returned if the current RedBoot installation does not have RBL support.

RBL Block Details

`rbl_get_block_details` can be used to get information about a specific RBL block, for example the primary code block. The function takes two arguments. The first identifies the particular RBL block that is of interest:

```

rbl_block_code_primary
rbl_block_code_backup
rbl_block_data_primary
rbl_block_data_backup

```

These identify an RBL block by purpose.

```

rbl_block_code_A
rbl_block_code_B
rbl_block_data_A
rbl_block_data_B

```

These identify an RBL block by memory location. RedBoot will allocate flash blocks to code and data in the above order.

The second argument should be a pointer to an `rbl_block_details` structure which will be used for storing the results.

```

typedef struct rbl_block_details {
    cyg_bool      rbl_valid;
    cyg_uint32    rbl_first_flash_block;
    void*         rbl_address;
    cyg_uint32    rbl_size;
    cyg_uint32    rbl_sequence_number;
} rbl_block_details;

```

At any time a particular RBL block may or may not contain valid code or data. For example if the system has an installed application which has not yet been updated then the primary code block will be valid but the backup code block will be invalid. The other fields will only be valid if the *rbl_valid* flag is set.

The *rbl_first_flash_block* and *rbl_address* fields give information about where an RBL block is held in memory. If an RBL block is spread over multiple flash blocks then care has to be taken: there may be one or more reserved flash blocks in the middle of an RBL block.

The *rbl_size* field gives the current size of a code or data block. This is the actual size specified when the block was updated, not the maximum size.

The *rbl_sequence_number* is used by the RBL code to distinguish between primary and backup blocks. It may also prove useful for debugging purposes.

`rbl_get_block_details` returns true on success, false on failure. The function can fail if the current RedBoot installation does not have RedBoot support, or if an invalid argument is passed.

Updating the Code

`rbl_update_code` is used to install a new code image. It takes two arguments, a buffer and a length. The buffer should contain an ELF executable valid for the target platform, usually stripped to remove unnecessary debug information. Filling this buffer is left to application code.

The function returns true on success, false on failure. It can fail if the current RedBoot installation does not have RedBoot support, if an invalid argument is passed, or if the specified size is larger than the flash space available for a code block.

A flash update involves erasing one or more flash blocks and then programming in the new data. It is important that while this is happening no other threads or interrupt handlers access the flash hardware since that would interfere with the update. Hence interrupts will be disabled for some time while the update is happening.

Updating the Data

`rbl_update_data` is used to install a new version of the persistent data. It takes two arguments, a buffer and a length. The RBL code does not care about the contents of the buffer, this is left entirely to application code.

The function returns true on success, false on failure. It can fail if the current RedBoot installation does not have RBL support, if an invalid argument is passed, or if the specified size is larger than the flash space available for a data block. It is also possible that support for persistent data was disabled completely inside RedBoot by setting the configuration option `CYGNUM_RBL_DATA_BLOCKS` to 0.

A flash update involves erasing one or more flash blocks and then programming in the new data. It is important that while this is happening no other threads or interrupt handlers access the flash hardware since that would interfere with the update. Hence interrupts will be disabled for some time while the update is happening.

Loading the Data

`rbl_load_data` is used to load the current version of the persistent data back into memory. It takes two arguments, a buffer and a length. If there is a valid primary data block then the RBL code will transfer that data into the specified buffer. The amount of data transferred will be either the actual block size or the length argument, whichever is smaller.

The function returns true on success, false on failure. It can fail if the current RedBoot installation does not have RBL support, if an invalid argument is passed, or if there is no current primary data block.

Restarting the Hardware

`rbl_reset` can be used to restart the hardware. Typically this is used after a code update. The function takes no arguments and does not return.

Application Library Extensions

Name

V2 RBL functions — allow applications to access RBL services

Synopsis

```
#include <cyg/rbl/rbl.h>

cyg_bool rbl_update_codeV(cyg_uint32 count, void* buffers[], cyg_uint32 lengths[]);
cyg_bool rbl_update_dataV(cyg_uint32 count, void* buffers[], cyg_uint32 lengths[]);
cyg_bool rbl_load_dataV(cyg_uint32 count, void* buffers[], cyg_uint32 lengths[]);
cyg_bool rbl_update_code_begin(void);
cyg_bool rbl_update_code_block(void* buffer, cyg_uint32 length);
cyg_bool rbl_update_code_end(void);
cyg_bool rbl_update_code_abort(void);
cyg_bool rbl_update_data_begin(void);
cyg_bool rbl_update_data_block(void* buffer, cyg_uint32 length);
cyg_bool rbl_update_data_end(void);
cyg_bool rbl_update_data_abort(void);
cyg_bool rbl_load_data_begin(void);
cyg_bool rbl_load_data_block(void* where, cyg_uint32 length);
cyg_bool rbl_update_data_end(void);
```

Description

The original V1 API required single buffers for all update and load operations. This proved unduly restrictive, especially when installing a new code image obtained over a network, because there would be no guarantee that a single buffer of the required size could be dynamically allocated when required. Hence the API was extended for V2 with vector functions, allowing the code and data to be spread over multiple buffers, and with transaction functions, allowing new code images to be installed a piece at a time.

Vector Functions

The three vector functions `rbl_update_codeV`, `rbl_update_dataV` and `rbl_load_dataV` work in terms of a series of buffers rather than a single buffer. For example `rbl_load_data` is equivalent to:

```
cyg_bool
rbl_load_data(cyg_uint8* where, cyg_uint32 size)
{
    cyg_uint8    dataV[1];
    cyg_uint32   sizesV[1];

    dataV[0] = where;
```

```

    sizesV[0] = size;
    return rbl_load_dataV(1, dataV, sizesV);
}

```

Obviously the vector functions become rather more useful for counts greater than 1. The update functions still require that all of the new images are resident in memory, but they no longer have to be in a single contiguous buffer.

When updating some flash drivers may impose limitations on the sizes. For example if the target hardware has a single 16-bit wide flash device then the flash driver may require that all flash write operations happen in multiples of 2 bytes, and the entries in the `sizesV` array should satisfy this requirement.

Transaction Functions

The transaction functions `begin/block/end` allow RBL operations to be performed in stages. For example `rbl_load_dataV` is equivalent to:

```

cyg_bool
rbl_load_dataV(cyg_uint32 count, cyg_uint8* whereV[], cyg_uint32 sizesV[])
{
    cyg_uint32 i;

    if (! rbl_load_data_begin()) {
        return false;
    }
    for (i = 0; i < count; i++) {
        if (! rbl_load_data_block(whereV[i], sizesV[i])) {
            return false;
        }
    }
    return rbl_load_data_end();
}

```

The `begin` function must be called at the start of a function. At any one time there can be only one code update, one data update, and one data load in progress, and the `begin` function will block if another thread is performing a conflicting RBL operation. Once the transaction is started the application can perform one or more block operations, and the transaction should normally be committed with an `end` function call. For an update the `begin` function will erase the appropriate flash blocks, the `block` function will write data to the flash, and the `end` function will write trailer data containing size, checksum and sequence number. At that point the image becomes the new primary code or data RBL block.

As an additional restriction, `rbl_update_data_end` will block if some other thread is currently loading data. This avoids confusion since otherwise that thread would end up loading data that is no longer primary, and it also avoids problems if another update operation is started immediately.

Unlike the simple or vector update functions, the transaction functions do not require that all of the new image is present in memory at the same time. Instead it is possible to begin a transaction, fetch the first part of the image over the network and install that, fetch the next part, and so on. If the application is unable to complete an update, for example because the network connection is lost, then there should be a call to the `abort` function instead of to

the end function. When a transaction is aborted no trailer gets written to flash so the new image remains invalid and the old image stays as the primary.

As with the vector operations the flash driver may impose limitations on the size arguments to `rbl_update_code_block` and `rbl_update_block`. For example if the target hardware uses a 16-bit wide flash chip then the size argument may have to be a multiple of two bytes.

Error Conditions

All of the RBL functions return a simple boolean to indicate failure. In reality failures are unlikely, but can be caused by the following:

1. The currently installed RedBoot was built without RBL functionality so there is no code in the system to keep track of RBL images and install new ones.
2. RedBoot uses a different version of the RBL protocol, for example V1 when the application has been built with V2.
3. The RedBoot RBL code was unable to initialize the system. This can happen if the actual hardware does not match the RedBoot configuration, for example if the flash chips actually present are smaller than expected and cannot hold all the code and data blocks specified by the `CYGDAT_RBL_RESERVED_FLASH_BLOCKS`, `CYGNUM_RBL_CODE_BLOCKS` and `CYGNUM_RBL_DATA_BLOCKS` configuration options.
4. An attempt is made to load or update data when RedBoot has been configured with zero RBL data blocks,
5. An attempt is made to load more data than is actually present in the current data image, or to install a new image that does not fit in the number of configured flash blocks.
6. An unexpected error occurs inside the flash driver, for example an attempt to erase a flash block fails.

For the transaction functions, if an error occurs then the transaction is automatically aborted. There is no need for the application to call `rbl_update_code_abort` or `rbl_update_data_abort` explicitly, or to end a load operation.

L. RedBoot Extra Initialization

RedBoot Extra Initialization

Name

CYGPKG_RBINIT — provide extra RedBoot initialization

Description

The RedBoot Extra Initialisation (RBINIT) package provides extra default initialization for RedBoot. This may be used to execute a set of initial commands, or to perform any additional platform or system specific initialization. The RBINIT package is aimed primarily at field deployment of production systems rather than at application development.

Building RedBoot

The first step in building RedBoot with RBINIT support is to create a RedBoot configuration appropriate for the target platform. This is somewhat target specific so the appropriate platform HAL or RedBoot documentation should be consulted for further details. Typically RedBoot should be configured for ROM startup.

Given this initial configuration the RBINIT package CYGPKG_RBINIT should now be added to the configuration, using one of the eCos configuration tools. For example with the command line tool this involves using **ecosconfig add rbinit**.

There are a number of other configuration options which may be changed at this point:

CYGGLO_RBINIT_STDIO_DISABLE

This option causes all output generated during the extra initialization to be discarded. Turn this option off to get output for debugging purposes.

CYGPKG_RBINIT_PRI

This option selects the priority of the extra initialization routine. The value of this option may either be RedBoot_INIT_BEFORE_NET or RedBoot_INIT_AFTER_NET which, as the names imply, cause the extra initialization to occur either before or after any network device is initialized.

Once RedBoot has been appropriately configured it can be built and installed as usual for the target platform.

Extra Initialization Function

The purpose of the extra initialization package is to execute the `rbinit_exec()` function. This function is at the end of the `rbinit.c` file.

The default content of this function provides support for loading and executing primary or secondary applications from a filesystem. The default function will first attempt to mount a JFFS2 filesystem named “jffs2” in the RedBoot FIS table. If that fails it will attempt to mount the first partition of an IDE hard disk using a FAT filesystem. Finally if that fails it will attempt to mount a RAM filesystem, although it will be empty. If a filesystem has been

successfully mounted, it will attempt to load into RAM and run a program named “`app.primary`” from the root of the filesystem, and if that fails, a program named “`app.secondary`”.

A customized version of this function may be made by either editing the file directly in the source repository, or by copying the file to the build tree and editing it there.

LI. ecoflash Flash Programming Utility

ecoflash Flash Programming Utility

Name

ecoflash — Flash Programming Utility

Synopsis

```
ecoflash --help [subcommand]
ecoflash [❶options] boards
ecoflash [❶options] info
ecoflash [❶options] program [-r | --raw] [-n | --no-erase] {file} [address]
ecoflash [❶options] dump [-a | --append] {file} [address] [length]
ecoflash [❶options] erase {address} [length]
ecoflash [❶options] write [-n | --no-erase] [-o offset | --offset=offset] {file} {address} [length]
ecoflash [❶options] lock {address} [length]
ecoflash [❶options] unlock {address} [length]
```

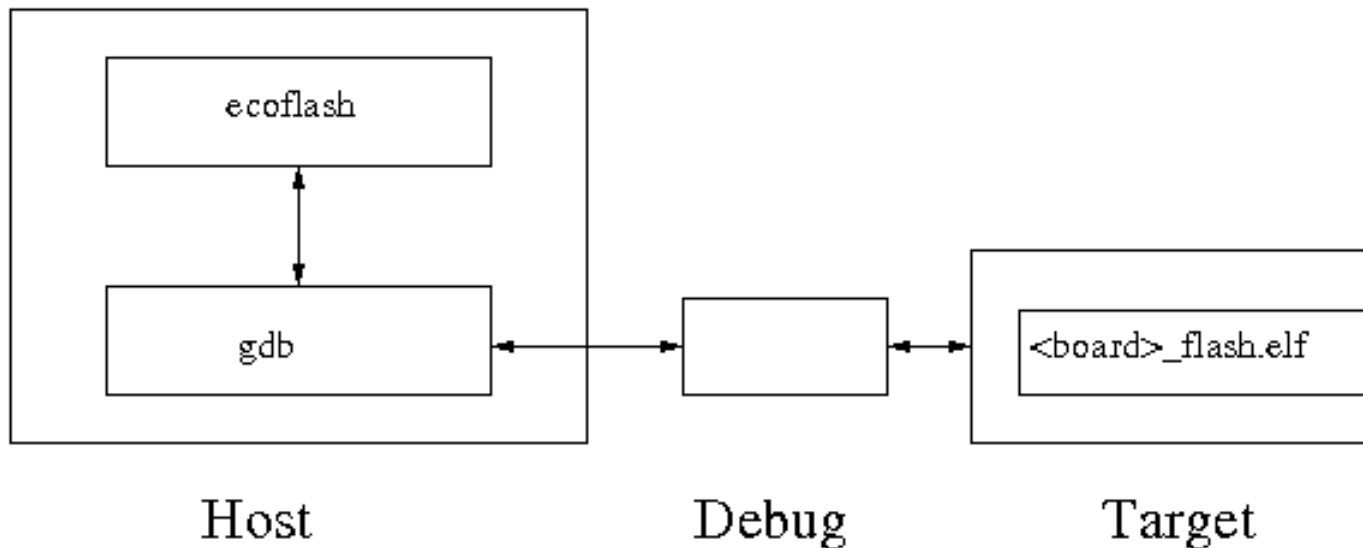
❶

```
[-h | --help]
[--version]
[--dry-run]
[-q | --quiet]
[-v | --verbose]
[-b board | --board=board]
[-g gdb_executable | --gdb=gdb_executable]
[-o objcopy_executable | --objcopy=objcopy_executable]
[-t target_command | --target=target_command]
```

Description

ecoflash is a utility for programming on-board flash via BDM, jtag, or similar hardware debug technology. There is nothing particularly original about this. Most manufacturers will provide similar utilities, and in fact those are likely to offer better performance because they operate at a lower level. The main advantages of ecoflash are: it provides a common user experience across a range of hardware; it is designed to work within a typical eCos development environment; and it has some built-in knowledge of how eCos systems work.

ecoflash works by running a suitable version of **gdb** on the host machine and running gdb commands. That version of gdb must either be able to drive the hardware debug technology directly, or more commonly it will in turn interact with some stub or daemon that knows how to drive the debug hardware. The target is initialized and a small target-side executable, for example `m5213evb_flash.elf`, is then loaded on to the target. The target-side executable is a simple eCos application that is linked with the eCos flash driver support, so it can be readily ported to any target for which a suitable flash driver is available. Manipulating the flash involves setting target-side variables used gdb commands, letting the target-side executable run until a breakpoint is hit, and then examining more target-side variables to determine the status.



The basic syntax is: `ecoflash [standard options] subcommand [options] args [optional args]`. The standard options provide information such as the target board, and most of these can be specified instead using environment variables. The subcommands specify the exact operation to be performed, for example to program an executable at the default location within the flash. Some subcommands take additional options, for example to suppress automatic erasure of flash blocks. These are followed by required arguments such as the executable filename, and possibly optional arguments. **ecoflash -h** with no additional argument provides help information for the utility as a whole. Additional information for a specific subcommand can be obtained using e.g. **ecoflash -h program**.

Standard Options

ecoflash accepts a number of standard options, for example the target board can be specified using `-b board`. These options will be used by several but not all of the subcommands.

`-h`

`--help`

Obtain help information about ecoflash or one of its subcommands.

`--version`

Display the ecoflash version string.

`--dry-run`

This suppresses the low-level block erase and write operations so the flash state does not actually change, but otherwise ecoflash operates normally including initializing the board, downloading the target-side executable, and having the latter initialize the eCos flash driver. It can be used to verify that the hardware is operating correctly.

-q
 --quiet
 -v
 --verbose

These can be used to reduce or increase the amount of feedback generated by ecoflash. -v can be specified several times for increased verbosity.

-b <board>
 --board=<board>

ecoflash needs to know which target board it should access, so that it can perform appropriate initialization and download the right target-side executable. **ecoflash boards** can be used to get a list of supported boards. It is possible to set an environment variable ECOFLASH_BOARD as an alternative to specifying this option on the command line every time:

```
$ export ECOFLASH_BOARD=m5213evb
```

The board name has two effects. It causes ecoflash to load a configuration file <board>.ecf with hardware-specific information, for example how to initialize the board using gdb commands. It also determines the target-side executable <board>_flash.elf. For both files ecoflash will first look in the current directory. If an ECOFLASH_DIR environment variable is defined then it will look in that directory. Finally it will look in the directory ../share/ecos/ecoflash relative to the ecoflash executable.

-g <gdb executable>
 --gdb=<gdb executable>

All subcommands except **boards** involve starting a gdb session on the host and downloading a target-side executable. Usually the gdb executable, for example **m68k-elf-gdb**, is specified in the board .ecf configuration file and found in the current PATH, so there is no need to use this option. However an alternative gdb can be specified if desired, for example when building and debugging an experimental version of gdb. The environment variable ECOFLASH_GDB can also be used instead of the command line option.

-o <objcopy executable>
 --objcopy=<objcopy executable>

The **program** subcommand can take an eCos executable in ELF format and automatically convert it to raw binary to program into flash. This involves running the appropriate host-side objcopy utility, for example **m68k-elf-objcopy**. By default ecoflash will munge the gdb file name to generate the objcopy name and will look for it on the PATH, so there is no need to use this option. An alternative objcopy can be specified on the command line or via the ECOFLASH_OBJCOPY environment variable.

-t <target command>
 --target=<target command>

ecoflash needs to know how to get gdb to interact with the debug hardware. The exact details of this depend not just on the hardware but also on the developer's setup, so cannot be provided by the board .ecf configuration file. Instead it must be provided by the user, in the form of a gdb target command. For example, if the debug hardware is accessed via a daemon on the local machine and that daemon listens on TCP/IP port 9000 for gdb remote protocol traffic then the gdb command to connect to the target hardware would be:

target remote localhost:9000. The `ecoflash -t` option should consist of everything after `target`, for example:

```
$ ecoflash -b alaia -t 'remote localhost:9000' info
```

Note the use of quote marks to make the shell treat it as a single argument, even though it contains spaces, and to prevent any processing of special characters like `$` and `|`. The `ECOFFLASH_TARGET` environment variable can be set instead:

```
$ export ECOFLASH_TARGET='remote localhost:9000'
```

Supported Boards

ecoflash boards can be used to get the names of the boards supported in the current installation, in other words what `-b` options are valid. A board is considered supported if `ecoflash` can find a `<board>.ecf` configuration file and a target-side executable `<board>_flash.elf`. It will search in the current directory, in the directory specified by the `ECOFFLASH_DIR` environment variable if that is defined, and in a directory relative to the `ecoflash` executable. For example if `ecoflash` is installed in `/usr/local/ecos/bin` then it will search in `/usr/local/ecos/share/ecos/ecoflash`.

Note that **ecoflash boards** only examines the file system and does not attempt to start `gdb` or interact in any way with target hardware.

Board Information

ecoflash info can be used to get information about a particular board, and also to verify that everything is working correctly.

```
% ecoflash -b m5213evb -t 'remote localhost:9000' info
Target board is m5213evb.
gdb is "m68k-elf-gdb", gdb target is "remote localhost:9000"
Target-side executable is version 1.
Detected 1 bank of flash.
  Start 0x00000000, end 0x0003ffff -> 256K.
    128 blocks of 2K.
Flash block locking is not supported.
Default program location for executables is 0x00000000.
Target-side buffer for read and write operation is 16K.
```

This shows the `gdb` command and the target string, which can be useful if that information comes from environment variables rather than the command line. `ecoflash` will run `gdb` and download the target-side executable, which reports itself as version 1. The target-side executable initializes the eCos flash driver and has detected the amount of flash reported and that lock and unlock operations are not supported. By default executables will be programmed at location `0x0`, and transfers between host and target use a 16k buffer (the M5213EVB only has a small amount of RAM, usually a larger buffer will be used).

Programming an Executable

ecoflash program can be used to install an eCos executable at the boot location within the flash. The executable should be linked against an eCos configuration with a suitable startup type, usually ROM or ROMRAM although this may vary between platforms. In its simplest form the subcommand just takes a single argument, a filename for the executable:

```
$ ecoflash program redboot.elf
Erasing 0x00000000 - 0x0000be57
Writing 0x00000000 - 0x00003fff (16384 bytes) from file "/tmp/redirect.1495", offset 0
Writing 0x00004000 - 0x00007fff (16384 bytes) from file "/tmp/redirect.1495", offset 16384
Writing 0x00008000 - 0x0000be57 (15960 bytes) from file "/tmp/redirect.1495", offset 32768
```

This assumes the `ECOFASH_BOARD` and `ECOFASH_TARGET` environment variables are set. **ecoflash** will examine the specified file. ELF executables will be automatically converted to a temporary raw binary file before being programmed into flash, using the `objcopy` utility. The default address within the flash is supplied by the target-side executable. Usually this will be the processor's boot location but the exact boot mechanism varies widely between processors and platforms.

ecoflash program takes two options. `-r` or `--raw` can be used to suppress the detection of ELF format files. Instead the file will be treated as a raw binary and no conversion is performed. This may be useful if the board has its own primary bootloader which expects to find an ELF executable at a particular address within the flash. `-n` or `--no-erase` can be used to suppress the automatic erase of the flash blocks prior to programming the flash. This may be useful on hardware where flash erase is optional, or if **ecoflash** is used in a batch environment where a previous step will have already erased the required amount of flash.

ecoflash program takes an optional additional argument, an alternative address within the flash. This may be useful if for example the board can boot from one of two locations depending on the state of a jumper.

Dumping Flash Contents

ecoflash dump can be used to read part or all of the flash and dump the data to a file on the host. This can be particularly useful when saving a known working image prior to replacing it with an experimental version. The default behaviour is to dump the entire flash contents:

```
$ ecoflash dump /tmp/working.bin
Dumping 0x00000000 - 0x00003fff (16384 bytes) to file "/tmp/working.bin"
Dumping 0x00004000 - 0x00007fff (16384 bytes) to file "/tmp/working.bin"
...
```

Optionally the starting address and the length can be specified:

```
$ ecoflash dump /tmp/working.bin 0xFFFF0000 128K
```

Lengths can be specified in bytes, kilobytes using a `K` suffix, megabytes using an `M` suffix, or flash blocks using a `B` suffix. Note that some flash devices have boot blocks of varying sizes so specifying a size in terms of blocks can be confusing.

ecoflash dump takes a single option, `-a` or `--append`. This causes ecoflash to append to the specified file instead of overwriting it.

Erasing Flash Blocks

ecoflash erase can be used to erase one or more flash blocks. This command is rarely needed since both the program and write subcommands will erase the required number of flash blocks by default, but may prove useful if the flash contains data other than an eCos executable and that data should be reset to uninitialized. In its simplest form the erase subcommand just takes an address:

```
$ ecoflash erase 0x40000
```

This causes ecoflash to erase the single flash block containing the specified address. Optionally a length can be specified, for example to erase 8 flash blocks:

```
$ ecoflash erase 0x40000 8B
```

The length can be specified in bytes, in kilobytes using a K suffix, in megabytes using an M suffix, or in flash blocks using a B suffix. Care must be taken if the specified address is not at the start of a flash block. For example if the address is 0x48000, the length is 128K, and flash blocks are 64K, then this is treated as a request to erase flash from 0x48000 to 0x67FFF. Since erase operations always involve whole flash blocks the actual erase affects all memory from 0x40000 to 0x6FFFF, so a total of 192K gets erased.

The erase subcommand does not have any options of its own, just the standard ones for all subcommands.

Writing Raw Data

ecoflash write can be used to write a raw data file to an arbitrary location within the flash. It is intended for installing additional data rather than the main executable, since the **program** subcommand is more appropriate for the latter. At least a filename and an address within the flash should be specified:

```
$ ecoflash write data.bin 0x00040000
Erasing 0x00040000 - 0x0004297c
Writing 0x00040000 - 0x0004297c (10621 bytes) from file "data.bin", offset 0
```

Optionally a length can be specified, for example:

```
$ ecoflash write data.bin 0x00040000 64K
```

This will write only the first 64K of data.bin rather than the whole file. The length can be specified in bytes, in kilobytes using a K suffix, in megabytes using an M suffix, or in flash blocks using a B suffix.

ecoflash write takes two options. `-n` or `--no-erase` can be used to suppress the automatic erase before the data is written to flash. This can be useful if a single flash block should contain data from more than one file: **ecoflash erase** would be used to erase the whole flash block, then two **ecoflash write -n** commands would program the two files

at the appropriate locations; alternatively the erase step can be skipped in subsumed by the first write, with only the second write using a `-n` option.

`-o <offset>` or `-offset=<offset>` can be used to skip part of a file. For example the following writes 12K of a file starting at a 4K offset:

```
$ ecoflash write --offset=4096 data.bin 12K
```

The offset can be specified in bytes, kilobytes using a K suffix, or megabytes using an M suffix.

Locking and Unlocking

On targets where the hardware and the flash driver support locking, the **lock** and **unlock** subcommands can be used to manipulate the locked status of one or more flash blocks. Both subcommands take an address and an optional length:

```
$ ecoflash lock 0x40000
...
$ ecoflash unlock 0x50000 256K
```

If no length is specified then just a single flash block is affected, unless the hardware implements locking at a coarser grain than individual flash blocks. The length can be specified in bytes, in kilobytes using a K suffix, in megabytes using an M suffix, or in flash blocks using a B suffix.

With some flash hardware locking is not persistent. Instead the locks are set to a default state when the flash chips are powered up or reset, usually locked. On such hardware the ecoflash lock and unlock subcommands are of little use since the locks would revert to their default state when ecoflash exits. Instead the target-side executable will either unlock all flash blocks during initialization or take whatever action is needed at run-time to handle erase and write operations.

Installation

Depending on your eCos distribution ecoflash may already be installed on your system. If not, installation is straightforward. The host-side consists of a single executable `ecoflash` in the package's `host` subdirectory. This is actually a platform-independent script written in the expect scripting language. It needs to be installed in a suitable location on the user's search `PATH`. The file can just be copied manually, or alternatively the `host` subdirectory contains a suitable configure script and support files:

```
$ <package path>/host/configure --prefix=/usr/local
$ make
$ make install
```

This will install ecoflash in the directory `/usr/local/bin`. Note that eCos also has a top-level configure script which will find subsidiary configure scripts inside the individual packages. A top-level configure/make/make install sequence will automatically install ecoflash as well as host-side support from other packages.

The ecoflash package contains only the generic support. It should be complemented by a `.ecf` configuration file and a `_flash.elf` target-side executable for every supported platform. The platform HAL's `misc` subdirectory usually holds a suitable `.ecf` file. The target-side executable will need to be rebuilt:

```
$ ecosconfig new <target> minimal
$ ecosconfig import <path>/ecoflash.ecm
$ ecosconfig tree
$ make
```

For an existing port there should be an `ecoflash.ecm` file in the platform HAL's `misc` subdirectory. Importing this will add the ecoflash package and any necessary support packages, set any platform-specific configuration options, and resolve any conflicts. After the make there should be a file `install/bin/flash.elf`, the target-side executable, and this should get renamed to `<board>_flash.elf` and installed in a location where ecoflash will find it.

Porting

Typically the only hard part of porting ecoflash to a new platform is to get gdb to interact with the jtag or BDM hardware and initialize the board. The porting process involves three steps: adding appropriate definitions to the platform HAL; building the target-side executable `<board>_flash.elf`; and writing the platform configuration file `<board>.ecf`.

The platform HAL must supply a single `#define`'d symbol corresponding to the default base address for **ecoflash program** operations. Usually this symbol gets defined in `cyg/hal/plf_io.h`, but the details may vary between architectures.

```
#define HAL_ECOFLASH_PROGRAM_BASE      0x00000000
```

Optionally the platform HAL can define a buffer size using `HAL_ECOFLASH_BUFLLEN`, an additional header file to include using `HAL_ECOFLASH_HEADER`, and an initialization macro `HAL_ECOFLASH_EXTRA_INIT()`. The latter may perform operations such as unlocking all flash blocks on hardware where locks are transient.

The target-side executable is a very simple eCos application that uses the generic flash driver support to interact with the hardware. Hence it assumes that a suitable flash driver is already available. Code and data sizes are both of the order of 4K, although obviously that will depend on the processor architecture. Usually the code will be RAM-resident and linked with a JTAG or RAM startup configuration. In addition a buffer is needed for transferring data between host and target. By default that buffer is 64K, corresponding to typical flash block sizes, but can be smaller if there is not enough RAM for a buffer that size. Building the target-side executable is straightforward:

```
$ ecosconfig new <target> minimal
$ ecosconfig add CYGPKG_IO_FLASH CYGPKG_LIBC_STRING CYGPKG_ECOFLASH
$ ecosconfig resolve
$ ecosconfig tree
$ make
```

It may be necessary to tweak the configuration data before generating the build tree, for example to change the startup type to JTAG or RAM. Adding the ecoflash package will result in one conflict related to the global compiler flags: by default ecoflash is built with no eCos debugging information, except for the ecoflash application itself.

The target-side executable may get checked into the source code control system as a binary, so avoiding debug information helps to keep the size down. Stripping out all debug information after the build is not possible because it would interfere with some of the gdb commands that ecoflash uses to interact with the target.

The result of the make is an executable `flash.elf` in the `install/bin` subdirectory. This should get renamed to `<target>_flash.elf` and installed to a directory where ecoflash will find it. Optionally an `ecoflash.ecm` file containing the configuration settings can be exported to facilitate future rebuilding.

Next it is necessary to write the configuration file `<target>.ecf`. This is a straightforward expect script that gets included by the main ecoflash executable. It should set variables `::gdb_executable` and `::command_prefix`. Optionally it may also define procedures `target_init0`, `target_init1`, and `target_kill`. `target_init0` is invoked after gdb has been started but before the `gdb target` command has been issued. `target_init1` is invoked after the `gdb target` command has been issued, and typically takes care of initializing the board via a sequence of gdb commands. To facilitate this the main ecoflash script provides procedures `gdb_run_command` and `gdb_run_quiet_command` which will do the hard work. `target_kill` is invoked just before shutting down the gdb session. The `<target>.ecf` file is typically placed in the platform HAL's `misc` subdirectory.

LII. Flash Safe

Flash Safe

Name

`CYGPKG_FLASHSAFE` — provide safe storage for data in flash memory

Description

The Flash Safe package provides a robust and simple mechanism for storing data in flash memory. It is intended for relatively small quantities of data such as configuration and customization data. For larger amounts of data, the JFFS2 flash filesystem is available.

The Flash Safe operates by dividing a region of the flash into a number of equal sized blocks. Each block is given a sequence number and is checksummed. A header contains these is stored at the start and end of each block. At startup the Flash Safe searches the available blocks for one with the latest sequence number and a valid checksum. The application can now retrieve data stored in the Flash Safe against a numeric key.

To store new data, the application opens a block, which will cause the block with the oldest sequence number to be erased and prepared for writing. Data can now be written to the block with a numeric key identifying each write. When all the data has been written, the block is committed, which will cause the headers to be written with valid checksums. This block now becomes the source of all subsequent data retrieval, freeing the original valid block for reuse.

This approach provides a simple transactional mechanism for storing data across power failures and crashes. At any time at least one committed block is valid and is released only after a new valid block has been committed to replace it. Any interruption during the creation of a new block will leave it invalid and data retrieval will fall back to the last committed block. The use of keys to identify data makes retrieval independent of the order in which the items are stored, of any change in size of the data and of any alignment requirements of the underlying flash device and driver.

The Flash Safe needs a minimum of two blocks to be defined, which must each be a multiple of the block size of the underlying flash device. A single Flash Safe block per flash device block would be the normal approach. More Flash Safe blocks may be used to implement a crude wear levelling mechanism, since under normal circumstances the Flash Safe will use the blocks in a round-robin manner.

Configuration

The flashsafe is mostly configured at runtime. The following CDL configuration options are present:

`CYGNUM_FLASHSAFE_BUFFER_SIZE`

This option defines the size of the buffer that the flashsafe uses to store data prior to writing it to disk. Different flash devices have different alignment and minimum sizes for writes to the flash. This buffer collects data items into segments that can be written in single operations.

`CYGPKG_FLASHSAFE_TESTS`

This lists the set of test programs. At present there is only one test, which runs on the synthetic target.

Flash Safe

Flash Safe Programmer Interface

Name

Flash Safe — API Details

Synopsis

```
#include <cyg/flashsafe/flashsafe.h>

int cyg_flashsafe_init(cyg_flashsafe *flashsafe);
int cyg_flashsafe_data(cyg_flashsafe *flashsafe, cyg_flashsafe_key key, void **data);
int cyg_flashsafe_read(cyg_flashsafe *flashsafe, cyg_flashsafe_key key, void *data,
cyg_uint32 *len);
int cyg_flashsafe_open(cyg_flashsafe *flashsafe);
int cyg_flashsafe_write(cyg_flashsafe *flashsafe, cyg_flashsafe_key key, void *data,
cyg_uint32 len);
int cyg_flashsafe_commit(cyg_flashsafe *flashsafe);
const char *cyg_flashsafe_errmsg(int err);
```

Description

The flash safe is accessed through this API. The flashsafe is initialized by calling `cyg_flashsafe_init()` passing it a pointer to a `cyg_flashsafe` structure. Within this structure the *base*, *block_count* and *block_size* fields must have been initialized to describe the area of flash to be managed. Typically, the structure would be defined statically as follows:

```
cyg_flashsafe flashsafe =
{
    .base           = 0x40000000,
    .block_count    = 3,
    .block_size     = 0x2000
};
```

If the flashsafe already contains data, then items may be retrieved using `cyg_flashsafe_data()`. The *key* argument identifies the data item to be retrieved. The *data* argument is a pointer to a location where a pointer to the data will be stored. Typically the pointer returned will refer directly to the flash device, and will thus be read-only. No size is returned, the application should either know what size the data is (e.g a structure or fixed sized array), or arrange for the data to be self-sized (e.g. a zero terminated string or contains a size field).

The function `cyg_flashsafe_read()` performs a similar job to `cyg_flashsafe_data()` except that the data is copied out of the flash memory. The *data* argument points to a buffer into which the data will be copied, and the *len* points to a location where the size of the buffer is stored when the call is made, and will contain the size of data copied in on return. This function is useful where it is intended to update the data read from the flashsafe,

possibly to write it back to flashsafe. `cyg_flashsafe_data()` is useful where the data only needs to be read, and saves valuable RAM space.

To open a flashsafe block for update the function `cyg_flashsafe_open()` should be called. This will select the oldest block in the safe, erase it and prepare it for writing.

The function `cyg_flashsafe_write()` is used to write new data to the current open block. The *key* argument should be an application selected 16 bit value that is used to identify this data item. This value should be unique for each item, otherwise the behaviour is undefined. The *data* and *len* arguments describe a buffer containing the data to be written.

The function `cyg_flashsafe_commit()` causes the current open block to be committed to the flash. This involves calculating the checksum and writing the header and trailer with the current sequence number.

Each of these functions may return one of a number of error codes. These may include the following:

CYG_FLASHSAFE_ERR_OK

This is returned when the operation succeeded.

CYG_FLASHSAFE_ERR_FLASH

This is returned when the flash device failed to initialize.

CYG_FLASHSAFE_ERR_MISMATCH

This is returned when there is a mismatch between the size any layout of the flashsafe described in the initialized `cyg_flashsafe` structure and the flashsafe found in flash.

CYG_FLASHSAFE_ERR_FLASH_PROG

This is returned when the flash driver reports a programming error.

CYG_FLASHSAFE_ERR_FLASH_ERASE

This is returned when the flash driver reports a block erase error.

CYG_FLASHSAFE_ERR_NO_DATA

This is returned when there is no current valid block in the flashsafe. This will usually only happen when the flashsafe is new.

CYG_FLASHSAFE_ERR_BAD_KEY

This is returned when a given key cannot be found in the flashsafe.

CYG_FLASHSAFE_ERR_NOT_OPEN

This is returned when a `cyg_flashsafe_write()` or `cyg_flashsafe_commit()` are called before making a call to `cyg_flashsafe_open()`.

CYG_FLASHSAFE_ERR_NO_SPACE

This is returned when the size of data is too large for either the buffer or the flashsafe block as a whole.

The function `cyg_flashsafe_errmsg()`, if given one of these error codes, will return a string describing the error.