YAMON™ Reference Manual

Document Number: MD00009
Revision 02.16
June 23, 2008

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies, Inc. All rights reserved.

Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Technologies, Inc. ("MIPS Technologies"). Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS Technologies or an authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines.

Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS TECHNOLOGIES, INC.

MIPS Technologies reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS Technologies or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government ("Government"), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or an authorized third party.

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS-3D, MIPS16, MIPS16e, MIPS32, MIPS64, MIPS-Based, MIPSsim, MIPSpro, MIPS Technologies logo, MIPS-VERIFIED, MIPS-VERIFIED logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4KSd, M4K, 5K, 5Kc, 5Kf, 24K, 24Kc, 24Kf, 24KE, 24KEc, 24KEf, 34K, 34Kc, 34Kf, 74K, 74Kc, 74Kf, 1004K, 1004Kc, 1004Kf, R3000, R4000, R5000, ASMACRO, Atlas, "At the core of the user experience.", BusBridge, Bus Navigator, CLAM, CorExtend, CoreFPGA, CoreLV, EC, FPGA View, FS2, FS2 FIRST SILICON SOLUTIONS logo, FS2 NAVIGATOR, HyperDebug, HyperJTAG, JALGO, Logic Navigator, Malta, MDMX, MED, MGB, OCI, PDtrace, the Pipeline, Pro Series, SEAD, SEAD-2, SmartMIPS, SOC-it, System Navigator, and YAMON are trademarks or registered trademarks of MIPS Technologies, Inc. in the United States and other countries.

All other trademarks referred to herein are the property of their respective owners.

Template: nB1.03, Built with tags: 2B

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table of Contents

Chapter 1: Introduction 7

Chapter 2: Getting Started 11
2.1: Installing YAMON 11
2.1.1: Environment 11
2.1.2: Unpacking 11
2.1.3: Building 12
2.2: Directory Hierarchy 12
2.3: Makefile Structure 14
2.4: Porting 16

Chapter 3: Modules 19
3.1: Summary of Modules 19
3.2: System Configuration (SYSCON module) 20
3.3: Driver API (IO Module) 21
3.3.1: Administrative Functions 22
3.3.2: Generic Driver Services 22

Chapter 4: Error Handling 25

Chapter 5: Cache Functions 29
5.1: Introduction 29
5.2: Cache Operations 29
5.3: Other Cache Issues 30
5.4: TLB 31

Chapter 6: Shell 33
6.1: Introduction 33
6.2: Shell Commands 35
6.2.1: Command Registration 35
6.2.2: Command Invocation 37
6.2.3: Error Reporting 37
6.2.4: Command Input/Output 37

Chapter 7: Exception Handling (EXCEP Module) 41
7.1: Overview 41
7.2: Reset Exception 42
7.3: NMI Exception 42
7.4: EJTAG Exception 43
7.5: Cache Error Exception 43
7.6: FPU Exception 43
7.7: Exception Handlers 43
7.7.1: exc_handler() 43
7.7.2: exception_sr() branch out 44
7.8: EXCEP API 45
7.9: Platform Adaptation of Interrupt Controller 46

Chapter 8: Applications 49
8.1: Loading Applications 49
8.2: Contexts 49
8.3: Go Command 49
8.3.1: Application API 50
8.3.2: GDB Command 51

Chapter 9: FPU Emulator 53
9.1: Overview 53
9.2: Support 53
9.3: Directory Structure 53
9.4: Cause Codes 54
9.5: 32/64-bit Stacking 54
9.6: API 54
9.7: Trampoline Code 55
9.8: How to Extract the Emulator 55
9.9: GPL-free YAMON build 56

Chapter 10: System Header Files 57

Chapter 11: Initialization 59

Chapter 12: PCI Configuration (PCI Module) 61
12.1: Introduction 61
12.2: Adaptation of pci_platform.c 61
12.2.1: pci_config() 61
12.2.2: arch_pci_system_slot() 62
12.2.3: arch_pci_slot() 63
12.2.4: arch_pci_slot_intline() 63
12.2.5: arch_pci_remote_intline() 63
12.3: Adaptation of pci_core.c 63
12.3.1: arch_pci_config_controller() 63
12.3.2: arch_pci_config_access() 63
12.3.3: arch_pci_lattim() 63
12.3.4: arch_pci_multi () 64

Chapter 13: CPU Reconfiguration 65
13.1: Overview 65
13.2: System Environment (SYSENV and ENV modules) 66

Appendix A: References 69

Appendix B: Revision History 71

List of Tables

Table 1.1: Supported Core Boards and CPUs 8
Table 2.1: YAMON Top Level Directories 13
Table 2.2: Architecture-specific Subdirectories 14
Table 2.3: Make all Resulting Files 16
Table 2.4: Make dis Resulting Files 16
Table 2.5: REVISION Register Layout 17
Table 2.6: 3rd Party REVISION Register Layout 17
Table 5.1: Cache Functions 30
Table 6.1: Command Line Recall/Editing Commands 34
Table 6.2: t_cmd Structure 36
Table 6.3: t_cmd_option Structure 36
Table 8.1: Initial Application Context 50
Table 9.1: FPU Emulation Directories 54

Chapter 1

Introduction

This document constitutes the Reference Manual for the YAMON™ ROM monitor, revision 02.16. It includes instructions on how to build a new version of YAMON, as well as a description of the structure of the source code of YAMON. This document supplements the *YAMON User's Manual* (Reference [1]) by including implementation details.

YAMON ("**Y**et **A**nother **MON**itor") is the ROM monitor used to control and monitor program execution on the evaluation and reference boards from MIPS Technologies Inc. YAMON includes boot code as well as traditional monitor functionality used for loading, executing, and debugging applications. YAMON source code is highly portable to other MIPS-based platforms. The latest version of YAMON can be downloaded from http://www.mips.com.

The target audience for this document is software designers who need detailed information on the structure of the YAMON source code, typically because they are adapting YAMON to new boards or CPUs, or they are copying parts of YAMON for use in other software projects.

YAMON actively supports the following boards:

- Malta™

In addition, YAMON currently supports the following boards, though support for these is not guaranteed in future releases:

- SEAD™
- SEAD-2™
- Atlas™

SEAD and SEAD-2 boards may be equipped with CPU cards that include one or more MIPS32® 4K® or MIPS64® 5K® class of CPUs. SEAD and SEAD-2 boards are shipped with a system controller ("Basic RTL") included for the on-board FPGA. YAMON supports "Basic RTL" revisions up to and including 01.03 and "MIPS SOC-it ® 101" system controller revision 1.1.

Malta and Atlas boards may be equipped with various "Core cards". A core card includes a CPU, a system controller (aka Northbridge) and an SDRAM module. Table 1.1 shows the core cards and CPUs supported by YAMON for the Malta and Atlas boards.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 1.1 Supported Core Boards and CPUs

	Board Types		
	CoreLV™	**CoreFPGA™**	**QED5261 Board™**
Available CPUs	MIPS32® 4K® class CPUs MIPS64® 5K® class CPUs	MIPS32® 4K® class CPUs MIPS64® 5K® class CPUs MIPS32® M4K™ (in FPGA)	QED RM5261®
System Controller	GT64120®		
Supported by Malta	Yes		
Supported by Atlas			
	Core20K™	**CoreBonito64™**	**CoreFPGA™ 2** **CoreFPGA™ 3** **CoreFPGA™ 4** **CoreFPGA™ 5**
Available CPUs	MIPS64® 20Kc™ MIPS64® 25Kf™	QED RM5261® QED RM7061A®	MIPS32® 4K™ class CPUs MIPS64® 5K™ class CPUs MIPS32® 24K® MIPS32® M4K™ MIPS32® 34K™ MIPS32® 74K™ MIPS32® 1004K™ (in FPGA)
System Controller	Bonito64		MIPS SOC-it® 101 MIPS ROC-it®
Supported by Malta	Yes		Yes
Supported by Atlas	No		Yes
	Core 24K®		
Available CPUs	MIPS32® 24K®		
System Controller	MIPS SOC-it®101		
Supported by Malts	Yes		
Supported by Atlas	Yes		

The same binary image is used for all boards and CPUs supported by YAMON. YAMON detects the specific board/ CPU at run-time.

The binary image contains both little- and big-endian code. YAMON detects the endianness at run-time and executes the appropriate code, and on RoHS-compliant Malta boards the endianness can be selected by hardware switch or via software setting.

YAMON is not an operating system. It does not support switching between several concurrent applications, nor does it manage the memory/TLB. YAMON is a monitor performing “bios” like system initialisation and allowing the user to examine/modify memory and memory mapped devices as well as loading and starting applications one at a time. As such, YAMON is well suited for hardware and software bringup.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Some of the main YAMON features are:

- System initialisation including RAM size/type detection and auto configuration, cache initialisation, PCI auto detection and auto configuration (PCI not available on all boards).
- Shell with command line history and editing.
- Traditional shell commands (`load`, `go`, `dump`, `edit`, etc.).
- Ethernet, IDE and serial port support (Ethernet and IDE not available on all boards).
- Configuration of CPU for CPUs supporting this.
- FPU emulation

YAMON supports the following interfaces:

- Command line interface through serial port.
- Debug interface through dedicated debug serial port. Interface conforms to GNU-GDB “Standard Remote Protocol” with extensions for SDE-GDB from MIPS Technologies Inc.
- Vector table based call interface for use by applications, PMON compatible.
- Ethernet for TFTP-load, save and “ping” support (Ethernet not available on all boards).
- IDE for reading and writing sectors on hard disk/compact flash (IDE not available on all boards).

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 2

Getting Started

This chapter describes how to install YAMON, the file structure used by YAMON, and how to port YAMON to a new board/CPU.

2.1 Installing YAMON

This section describes how to unpack and build the YAMON software.

2.1.1 Environment

YAMON has been built and tested in the following environment:

- Host running Sun Sparc Solaris 2.7 or Red Hat Linux release 7.0.
- GNU Make version 3.77.
- GNU compiler tools (gcc, ld, objcopy, objdump):
  Cygnus GNUPro Embedded ToolSuite with MIPS support (e.g., gcc-2.9) or
  Algorithmics SDE-MIPS 4.0b or
  Algorithmics SDE-MIPS 4.1
  MIPS Toolkit
- GNU Perl v5.6.0
- GNU gunzip tool.

2.1.2 Unpacking

YAMON source code may be distributed on CD without being compressed in any way. It may also be distributed in gzipped tar files with the following naming convention:

```
yamon-src-<rev>.tar.gz
```

The compressed file should be moved to the directory, where you wish YAMON to be installed, and the following operations should be performed (<*rev*> is assumed to be 02.16):

```
gunzip yamon-src-02.16.tar.gz
tar xvf yamon-src-02.16.tar
```

This procedure will create a directory named "`yamon`" containing the source tree of YAMON.

The base of the YAMON source tree is termed $(`ROOT`) in the remainder of this document. All references to files or directories are based at $(`ROOT`).

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

2.1.3 Building

This section describes how to build the YAMON image. See Section 2.3 "Makefile Structure" for a full description of the makefile structure and make targets.

YAMON is written in C and assembler code. Any ANSI-C compliant compiler should be able to build YAMON. However, the makefile assumes GNU GCC compiler tools.

The main makefile is named:

```
bin/Makefile
```

A separate makefile is invoked by the main makefile for building the FPU emulator:

```
fpuemul/Makefile
```

The main makefile contains the following expressions, which define the names of the compiler tools (replace this according to your setup).

```
# Tool-chain used for compilation of target code (cygnus, sde or linux).
#TOOLCHAIN = cygnus
TOOLCHAIN = sde
#TOOLCHAIN = linux
```

YAMON is built from the `bin` directory by issuing the commands:

```
make install
make
```

`make install` simply generates directories `bin/EL, bin/EB, ../fpuemul/EL and ../fpuemul/EB` used for holding little-endian and big-endian object code.

`make` will generate the following files:

```
yamon-<rev>.fl
yamon-<rev>.bin
```

`yamon-<rev>.fl` contains the YAMON image in the format required for programming the Monitor flash memory using the parallel port or USB (depending on platform).

`yamon-<rev>.bin` contains the YAMON image in binary format suitable for (E)PROM programming hardware.

See Ref[1] for instructions on how to upgrade your specific board with the YAMON image.

2.2 Directory Hierarchy

The first level of the YAMON source tree located at $(`ROOT`) contains the subdirectories listed in Table 2.1.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 2.1 YAMON Top Level Directories

Name	Short description
`bin`	Makefile, linker scripts and binary images. See Section 2.3 "Makefile Structure".
`include`	Architecture independent header files.
`fpuemul`	Architecture independent floating-point emulation code. See Chapter 9, "FPU Emulator" on page 53.
`init`	Architecture independent initialisation code. See Chapter 11, "Initialization" on page 59.
`arch`	Architecture-specific code. See Table 2.2.
`shell`	Architecture independent part of YAMON shell and shell commands. See Chapter 6, "Shell" on page 33.
`sys`	Architecture independent system functions used throughout YAMON.
`load`	Functions for loading and interpreting s-records.
`net`	NET module (network stack including arp, udp, tftp, ping etc.)
`drivers`	Device drivers.
`io`	IO module (Driver interface). See Section 3.3 "Driver API (IO Module)".
`lib`	Standard library.
`sysenv`	SYSENV (System environment) module (handling records kept in flash). See Section 13.2 "System Environment (SYSENV and ENV modules)".
`env`	Architecture independent part of ENV Module (handling environment variables). See Section 13.2 "System Environment (SYSENV and ENV modules)".
`exception`	Architecture independent part of EXC module (exception handling). See Chapter 7, "Exception Handling (EXCEP Module)" on page 41.
`pci`	Architecture independent part of PCI module (PCI auto detection and auto configuration). See Chapter 12, "PCI Configuration (PCI Module)" on page 61.
`syscon`	Architecture independent part of SYSCON module (System Configuration). See Section 3.2 "System Configuration (SYSCON module)".

Header files are generally located in the `include` and `arch/include` directories. A few local header files are located in the directory of the corresponding module. Public header files for modules generally contain the postfix "`api`". For example, the EXCEP interface is defined in the header file `include/excep_api.h`.

Architecture (board/CPU) specific source code is in the "`arch`" directory. Ideally, when porting YAMON to a new board/CPU, this is the only place where modifications should be made (except for new drivers, which belong in the "driver" directory). See also Section 2.4 "Porting".

The "`arch`" directory contains the subdirectories listed in Table 2.2.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 2.2 Architecture-specific Subdirectories

Name	Description
include	Architecture-specific header files.
reset	Architecture-specific reset code and early initialisation code executed from flash.
init	Architecture-specific initialisation code.
pci	Architecture-specific part of PCI module.
shell	Architecture-specific shell code.
syscon	Architecture-specific part of SYSCON module.
sys	Architecture-specific system functions used throughout YAMON.
env	Architecture-specific part of ENV module.
exception	Architecture-specific part of EXCEP module.
freq	FREQ module (detection of CPU and bus frequencies).
isa	ISA module (ISA address mapping).

2.3 Makefile Structure

The makefile structure is described in this section.

All assembler files are expected to be named <*name*>.S rather than <*name*>.s (i.e., use capital 'S') since, by default, the C-preprocessor is only run on '.S' files.

All C files are expected to be named <*name*>.c.

Making YAMON actually involves making three software images, which are then concatenated to a single image. The images are:

- Start up code that determines endianness and passes control to either the little- or big-endian image.
- Little-endian image.
- Big-endian image.

There is one main makefile for YAMON. It is named:

```
bin/Makefile
```

Header files used by YAMON are part of the YAMON source code. There is only one exception to this, which is the <stdarg.h> header file for passing variable sized argument lists. Because this file is tightly dependent on the compiler's calling convention (and not an operating system), it must be supplied by the toolchain.

There are several linker scripts involved in making YAMON:

```
bin/link/link.xn
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
bin/link/link_el.xn
bin/link/link_eb.xn
```

The first linker script is used for the start up code, while the last two contain the layout of the little-endian and big-endian code.

Besides these linker scripts, there are three linker scripts used when generating code suitable for use on a hardware simulator. Modifications include linking all code and initialized data to RAM, except the very first code located at the reset vector. This way, code and initialized data need not be copied from flash/(E)PROM to RAM.

When building an image for use on a hardware simulator, we also assume that the caches need not be initialized and RAM need not be cleared. Avoiding these time-consuming tasks is a big benefit when simulating the hardware. The symbol `_SIMULATE_` is defined by the makefile and used for conditional compilation of relevant parts of the code.

The linker scripts used for this code are:

```
bin/link/link_sim.xn
bin/link/link_el_sim.xn
bin/link/link_eb_sim.xn
```

The make procedure requires a Perl script for conversion of formats:

```
bin/tools/srecconv.pl
```

`srecconv.pl` converts an s-record file to two files: A file with the image in the format required for download to flash using the parallel port or USB, and a binary file in the format required for programming an (E)PROM device.

The file used as input file for `srecconv.pl` (`yamon-<`*`rev`*`>.rec`) is not a “normal” s-record file. It is created by the makefile by concatenating the 3 images (start up code, little endian code and big endian code). Also, the special strings “`!B`” and “`!L`” have been inserted in `yamon-<`*`rev`*`>.rec` in order to notify `srecconv.pl` about the endianness of the 3 sections.

The following targets are available for make:

- `install` generates subdirectories used for the little- and big-endian object files.
- `clean` deletes the files generated by make.
- `depend` generates dependencies.
- `all` (default) builds the images.
- `dis` generates disassembly files.

The following should be noted:

- `make install` should be run before making other targets.
- `make clean` deletes dependency files and “make all” does not generate dependency files. So, “`make clean`” should be followed by “`make depend`” in case dependency files are wanted.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

The final files generated by `make all` are listed in Table 2.3.

Table 2.3 Make all Resulting Files

`bin/yamon-<rev>.fl`	YAMON image in format required for download to flash using parallel port or USB.
`bin/yamon-<rev>.bin`	YAMON image in binary format suitable for (E)PROM programming devices.
`bin/reset-<rev>.map`	Linker generated map file for the startup code located at the reset exception vector.
`bin/EL/yamon-<rev>_el.map`	Linker generated map file for the little endian image.
`bin/EB/yamon-<rev>_eb.map`	Linker generated map file for the big endian image.

The files generated by `make dis` are listed in Table 2.4.

Table 2.4 Make dis Resulting Files

`bin/reset-<rev>.dis`	Disassembly of startup code located at the reset exception vector.
`bin/EL/yamon-<rev>_el.dis`	Disassembly for the little endian image.
`bin/EB/yamon-<rev>_eb.dis`	Disassembly for the big endian image.

2.4 Porting

Porting YAMON to a new board/CPU is done by adapting the files in the '`arch`' directory. New device drivers belong in the '`drivers`' directory.

Architecture-specific directories may be further split in subdirectories:

- `cpu`: CPU-specific code.
- `platform`: Platform-specific code.
- `platform/core`: "Core card" specific code. "Core cards" are used on the Atlas and Malta boards. They hold the CPU, the system controller (Northbridge), and the SDRAM module. `platform/core` holds the system controller specific code. For some platforms, it may make more sense to place this code in the `platform` directory.

In order to port YAMON to a new platform, the recommended practice is to go through the files of the relevant directories (e.g., `arch/<subdir>/cpu` when a new CPU is added). The functions are commented, and switch constructs have been used extensively for board/CPU selection.

Also, it is strongly recommended to read this document, but you are obviously doing that already. For example, adaptation of interrupt handling is described in Section 7.9 "Platform Adaptation of Interrupt Controller" and adaptation of PCI configuration is described in Section 12.2 "Adaptation of pci_platform.c" and Section 12.3 "Adaptation of pci_core.c".

YAMON code is to some extent arranged in modules, with indirectly-called functions. Porting may involve adding new modules or removing modules. See Chapter 3, "Modules" on page 19for information about how to do this.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

All boards supplied by MIPS Technologies use the memory- mapped location MIPS_REVISION (0x1fc00010) for the *REVISION* register used for board identification. The format of the *REVISON* register is shown in Table 2.5.

Table 2.5 REVISION Register Layout

Bits	Field Name	Function
31:8	Reserved	Board-specific extensions.
7:4	*PROID*	Identifies the basic system (motherboard) type
3:0	*PRORV*	Identifies the basic system (motherboard) revision

Throughout the YAMON arch directory, board-specific code is executed based on the global variable:

```
sys_platform
```

sys_platform holds the value of the PROID field.

The values available for sys_platform are defined in the file:

```
arch/include/product.h
```

In this version of YAMON, the defined values are:

```
/* Encoding of proid field */
#define PRODUCT_ATLASA_ID       0x0   /* Atlas board   */
#define PRODUCT_SEAD_ID         0x1   /* SEAD board    */
#define PRODUCT_MALTA_ID        0x2   /* Malta board   */
#define PRODUCT_SEAD2_ID        0x3   /* SEAD-2 */
#define PRODUCT_THIRD_PARTY_ID  0xe   /* Third party */
```

The value 0xe (PRODUCT_THIRD_PARTY_ID) is allocated for third-party products ("Non MIPS Technologies products"). The layout of bits 31:8 are different for "Non MIPS Technologies products," as shown in Table 2.6.

Table 2.6 3rd Party REVISION Register Layout

Bits	Field Name	Function	Initial Value
31:24	Reserved		0x0
23:16	*MANPD*	Manufacturer's product ID	-
15:8	*MANID*	Manufacturer's ID code	-
7:4	*PROID*	Basic system type	0xE
3:0	*PRORV*	Product revision	-

The specific third-party product is identified by the *MANID* / *MANPD* fields.

The *MANID* field is allocated by MIPS Technologies, while *MANPD* is allocated by the company.

A platform need not necessarily support the *REVISION* register, but the YAMON port must set the following variables:

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

- `sys_platform` (should be set to 0xE).
- `sys_manid` (allocated by MIPS Technologies).
- `sys_manpd` (allocated by company that owns the board).

An additional variable `sys_corecard` is used for the Atlas and Malta boards to identify the "Core card" (plug-on card holding CPU, system controller, and SDRAM).

These variables are set in function `arch_platform_init`() in the file:

```
arch/init/platform/init_platform.c
```

The *`REVISION`* register is also accessed in the following files, which must therefore be adapted:

```
arch/reset/bootvector/reset.S
arch/reset/init_platform_s.S
arch/reset/init_core_s.S
arch/syscon/platform/syscon_platform.c
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 3

Modules

YAMON operates with “Modules”. The modules to be included (at compile time) in YAMON’s start up sequence are defined in the header file `arch/include/initswitch.h`.

3.1 Summary of Modules

All modules listed below are included in this Release.

- SYSCON module (System Configuration). See Section 3.2 “System Configuration (SYSCON module)”.
- IO module (Driver interface). See Section 3.3 “Driver API (IO Module)”.
- EXCEP module (Exception handling). See Chapter 7, “Exception Handling (EXCEP Module)” on page 41.
- RTC module (RTC driver).
- FREQ module (Frequency detection).
- PCI (PCI auto detection and auto configuration). See Chapter 12, “PCI Configuration (PCI Module)” on page 61.
- IIC (IIC drivers).
- EEPROM_IIC (Driver for NM24C09 devices).
- FLASH_STRATA (Driver for flash devices conforming to the “Common Flash Interface” (CFI) specification).
- SYSENV module (Manages the system flash). Section 13.2 “System Environment (SYSENV and ENV modules)”.
- ENV (manages environment variables). See Section 13.2 “System Environment (SYSENV and ENV modules)”.
- SERIAL (UART drivers).
- LAN_SAA9730 (Driver for SAA9730 Ethernet controller).
- LAN_AM79C973 (Driver for AM79C973 Ethernet controller).
- NET (Networking protocols).
- IDE (IDE support).

The list shown above corresponds to the following code in `initswitch.h`:

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
/* switches to control which modules to install in YAMON */
#define  INCLUDE_SYSCON             1
#define  INCLUDE_IO                 1
#define  INCLUDE_EXCEP              1
#define  INCLUDE_RTC                1
#define  INCLUDE_FREQ               1
#define  INCLUDE_PCI                1
#define  INCLUDE_IIC                1
#define  INCLUDE_EEPROM_IIC         1
#define  INCLUDE_FLASH_STRATA       1
#define  INCLUDE_SYSENV             1
#define  INCLUDE_ENV                1
#define  INCLUDE_SERIAL             1
#define  INCLUDE_LAN_SAA9730        1
#define  INCLUDE_LAN_AM79C973       1
#define  INCLUDE_NET                1
#define  INCLUDE_IDE                1
```

The sequence of module initialisation is important because some modules depend on others. Therefore, changing the sequence should be done with great care.

Modules are initialised by the function `initmodules`() in file:

```
arch/init/platform/initmodules.c.
```

`initmodules`() is called during initialisation of YAMON (see Section 17, "Initialisation Code").

3.2 System Configuration (SYSCON module)

YAMON is designed to determine the hardware platform (board and CPU) at run time and configure itself accordingly.

Many parts of the code, in particular the shell and the drivers, do not have any information about the specific platform, but acquire platform-specific data (for example, the memory mapping of devices, serial port data, etc.) and other centralized system data through a “System Configuration” (SYSCON) module.

SYSCON is divided into an architecture-independent part, implemented in the files:

```
syscon/syscon.c
syscon/syscon_tty.c
```

and an architecture-specific part, implemented in the files:

```
arch/syscon/cpu/syscon_cpu.c
arch/syscon/platform/syscon_platform.c
arch/syscon/platform/syscon_platform_tty.c
arch/syscon/platform/core/syscon_core.c
```

The SYSCON API is defined in the header file

```
arch/include/syscon_api.h
```

The SYSCON API defines the following interface functions:

```
INT32
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
SYSCON_init( void )

INT32
SYSCON_read(
    UINT32 param_id,     /* IN: one of the 'SYSCON_xyz_ID' param id's */
    void   *param,       /* INOUT: parameter value                    */
    UINT32 param_size)   /* IN:    parameter size (bytes)             */

INT32
SYSCON_write(
    UINT32 param_id,     /* IN: one of the 'SYSCON_xyz_ID' param id's */
    void   *param,       /* IN: parameter value                       */
    UINT32 param_size)   /* IN: parameter size (bytes)                */
```

SYSCON_init() is called during YAMON initialisation. It initializes the internal data structures of the SYSCON module.

SYSCON_read() and SYSCON_write() functions are called whenever software modules need to access system parameters. Each parameter is identified by a unique ID defined in the SYSCON API header file.

SYSCON objects may be read through the application interface (see Section 8.3.1 "Application API"). In order to retain backward compatibility with YAMON 02.00, IDs used for YAMON 02.00 SYSCON objects were kept unmodified in this version of YAMON. New object IDs are always allocated after IDs from previous versions.

3.3 Driver API (IO Module)

YAMON's access to supplied hardware is organized into a number of drivers with a common API. Currently, the following drivers are available:

- EEPROM driver for NM24C0 device.
- IIC driver for SAA9730 device.
- IIC driver for PIIX4 Southbridge device.
- IIC driver for SEAD (Basic RTL and MIPS SOC-it 101).
- Driver for flash devices conforming to the "Common Flash Interface" (CFI) specification.
- LAN driver for SAA9730 Ethernet device.
- LAN driver for AM79C973 Ethernet device.
- Real Time Clock driver for DS1687 compatible devices.
- UART driver for SAA9730 UART device.
- UART driver for TI16550 compatible UART devices.
- IDE driver for PIIX4 South bridge device.

All drivers are accessed through an Input/Output (IO) module implemented in the file:

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
io/io.c
```

The API is defined in the file

```
include/io_api.h
```

The API defines interface functions, which can be grouped into:

- Administrative functions, and
- Generic driver services

Each of the above is described in the following subsections.

Drivers are identified by major and minor device numbers. Major device numbers are defined in:

```
include/sysdev.h
```

Minor numbers are defined in specific <*driver*>_api.h files.

3.3.1 Administrative Functions

Administrative functions are used during initialization to allocate table space and install the drivers needed:

```
INT32 IO_setup(
    UINT32 devices  ) ; /* IN: max. 'devices' to be supported     */


INT32 IO_install(
    UINT32       major,    /* IN: major device number             */
    t_io_service init,     /* 'init'  service function pointer    */
    t_io_service open,     /* 'open'  service function pointer    */
    t_io_service close,    /* 'close' service function pointer    */
    t_io_service read,     /* 'read'  service function pointer    */
    t_io_service write,    /* 'write' service function pointer    */
    t_io_service ctrl ) ;  /* 'ctrl'  service function pointer    */
```

3.3.2 Generic Driver Services

Generic driver services are used to request services provided by installed drivers (the classic set of '`init`', '`open`', '`close`', '`read`', '`write`', '`ctrl`'):

```
INT32 IO_init(
    UINT32 major,          /* IN: major device number             */
    UINT32 minor,          /* IN: minor device number             */
    void   *p_param ) ;    /* INOUT: device parameter block       */

INT32 IO_open(
    UINT32 major,          /* IN: major device number             */
    UINT32 minor,          /* IN: minor device number             */
    void   *p_param ) ;    /* INOUT: device parameter block       */

INT32 IO_close(
    UINT32 major,          /* IN: major device number             */
    UINT32 minor,          /* IN: minor device number             */
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
    void   *p_param ) ;    /* INOUT: device parameter block       */

INT32 IO_read(
    UINT32 major,          /* IN: major device number             */
    UINT32 minor,          /* IN: minor device number             */
    void   *p_param ) ;    /* INOUT: device parameter block       */

INT32 IO_write(
    UINT32 major,          /* IN: major device number             */
    UINT32 minor,          /* IN: minor device number             */
    void   *p_param ) ;    /* INOUT: device parameter block       */

INT32 IO_ctrl(
    UINT32 major,          /* IN: major device number             */
    UINT32 minor,          /* IN: minor device number             */
    void   *p_param ) ;    /* INOUT: device parameter block       */
```

A specific driver may choose to implement a subset of the 6 generic services fitting the required services of that particular driver. Calling a service that is not provided is handled by the IO module, and an error code is returned.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 4

Error Handling

This chapter describes how errors are handled by YAMON.

YAMON provides a mechanism for handling errors in a consistent manner. Modules are allocated a domain (range of 0x1000 values) defined in the file:

```
include/syserror.h
```

Error numbers are allocated as follows:

```
*      ERROR CODE RANGE          MODULE              DEFINED IN       *
*    =====================   ==================    ==============    *
*    0000'0001 - 0000'0FFF   (reserved)                                *
*    0000'1000 - 0000'1FFF   (reserved)                                *
*    0000'2000 - 0000'2FFF   IO                    io_api.h            *
*    0000'3000 - 0000'3FFF   SERIAL                serial_api.h        *
*    0000'4000 - 0000'4FFF   LAN                   lan_api.h           *
*    0000'5000 - 0000'5FFF   IIC                   iic_api.h           *
*    0000'6000 - 0000'6FFF   EEPROM                eeprom_api.h        *
*    0000'7000 - 0000'7FFF   RTC                   rtc_api.h           *
*    0000'8000 - 0000'8FFF   SYSCON                syscon_api.h        *
*    0000'9000 - 0000'9FFF   FLASH                 flash_api.h         *
*    0000'A000 - 0000'AFFF   NET                   net_api.h           *
*    0000'B000 - 0000'BFFF   EXCEP                 excep_api.h         *
*    0000'C000 - 0000'CFFF   SYSENV                sysenv_api.h        *
*    0000'D000 - 0000'DFFF   LOADER                loader_api.h        *
*    0000'E000 - 0000'EFFF   ENV                   env_api.h           *
*    0000'F000 - 0000'FFFF   PCI                   pci_api.h           *
*    0001'0000 - 0001'0FFF   SHELL                 shell_api.h         *
*    0001'1000 - 0001'1FFF   IDE                   ide_api.h           *
```

`syserror.h` also defines the following two generic error codes:

```
#define  OK             0x00000000
#define  NOT_OK         0xffffffff
```

Domain numbers are found by right-shifting error numbers by 12 bits. For example, the domain number for the LAN module is 0x4. Actually, there are two LAN modules in YAMON, since there are two different Ethernet drivers. However, only one will be installed on any particular platform, so they share the error range.

The range 0x000 to 0x0FFF is reserved for non-unique error codes bypassing the global error handling system. This range is not used in this version of YAMON.

A module registers an error lookup function to SYSCON. This function converts a specific error number within the module domain to text strings describing the error.

The SYSCON ID used for registering an error lookup function is:

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
SYSCON_ERROR_REGISTER_LOOKUP_ID
```

The following code, from `sysenv/sysenv.c`, illustrates the registration mechanism.

```
t_sys_error_lookup_registration registration;

/* register lookup syserror */
registration.prefix = SYSERROR_DOMAIN( ERROR_SYSENV );
registration.lookup = SYSENV_error_lookup;
SYSCON_write( SYSCON_ERROR_REGISTER_LOOKUP_ID,
          &registration,
          sizeof( registration ) );
```

The lookup function has the following format (from `syscon_api.h`):

```
typedef INT32 (*t_sys_error_lookup)(
    t_sys_error_string *p_param ) ; /* INOUT: code to string(s)   */
```

The input/output parameter is a pointer to a structure of type `t_sys_error_string`, defined as follows (from `syscon_api.h`):

```
typedef struct sys_error_string
{
    UINT32 syserror ;  /* system error code to be converted           */
    UINT32 count ;     /* number of string-pointers returned          */
    UINT8  **strings ; /* pointer to array of string-pointers         */
} t_sys_error_string ;
#define SYSCON_ERRORMSG_IDX 0 /* String index for error message       */
#define SYSCON_DIAGMSG_IDX  1 /* String index for diagnose message    */
#define SYSCON_HINTMSG_IDX  2 /* String index for hint message        */
```

Up to 3 strings may describe the error:

- The primary error message.
- A diagnostic message.
- A hint message.

Whenever an error indication defined by a global error number is received, SYSCON may be requested to lookup the error strings (which need not be static). In this version of YAMON, the only module performing error lookup is the shell.

The SYSCON object used for this is:

```
SYSCON_ERROR_LOOKUP_ID
```

The lookup is performed in the file `shell/shell.c`:

```
t_sys_error_string error_string ;

/* lookup syserror */
error_string.syserror = err;
error_string.count    = MAX_NUMBER_OF_ERROR_STRINGS ;
error_string.strings  = err_strings ;
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
for (i=0; i<3; i++)
{
    error_string.strings[i] = NULL ;
}

SYSCON_read( SYSCON_ERROR_LOOKUP_ID,
             &error_string,
             sizeof( error_string ) );
```

SYSCON will direct the call to the error-lookup function registered by the specific module. This error-lookup function then writes the `error_string` structure.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 5

Cache Functions

This chapter describes the cache operations supported by YAMON.

5.1 Introduction

The set of cache functions listed in Table 5.1 are available. They are implemented and defined in the following files:

```
sys/cpu/cpu.c
sys/cpu/cpu_s.S
include/sys_api.h
```

The functions also support a level 2 cache, if one is available and enabled. Presence of an L2 cache is by default disabled, but may be enabled using the shell `cache` command (which causes the function `sys_cpu_l2_enable`() to be called).

5.2 Cache Operations

Cache operations may address cache lines of the I- or D- cache in one of the following ways:

- All cache lines
- A specific cache line using the index of the line
- The cache line (if any) that references a particular virtual address (may correspond to one of several cache lines, depending on cache associativity)

The index of a cache line includes the cache way as well as the line within that way, as defined by the MIPS "`cache`" instruction.

Indexed I-cache operations always invalidate the requested line based on the index of the line.

Indexed D-cache operations will write the contents of the cache line to memory, if the line is valid and dirty. The cache line will be set to the invalid state.

Addressed I-cache operations will perform the same operation as the indexed I-cache operation on the line (if any) containing the virtual address.

Addressed D-cache operations will perform the same operations as the indexed D-cache operation on the line (if any) containing the virtual address.

Note that the functions `sys_icache_invalidate_addr`() and `sys_icache_invalidate_all`() also flush the CPU pipeline by performing an `eret` instruction (actually performed by the function `sys_flush_pipeline`()). This is required because instructions in the cache line being invalidated may already be in the CPU pipeline.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 5.1 Cache Functions

Function	Description
`void sys_icache_invalidate_index( UINT32 index );`	Invalidate I-cache line at location defined by '`index`'.
`void sys_icache_invalidate_addr( UINT32 address );`	If any I-cache line contains '`address`', invalidate the line. This function also flushes the pipeline.
`void sys_icache_invalidate_all( void );`	Invalidate entire I-cache and flush the pipeline.
`void sys_dcache_flush_index( UINT32 index );`	If the D-cache line at location defined by '`index`' is valid and dirty, write the line to memory. Invalidate the line.
`void sys_dcache_flush_addr( UINT32 addr );`	If any D-cache line contains '`address`', perform the following : If the line is valid and dirty, write it to memory. Invalidate line.
`void sys_scache_flush_index( UINT32 index );`	Flush L2 cache line containing specified index.
`void sys_dcache_flush_all( void );`	Write all valid and dirty D-cache lines to memory. Invalidate entire D-cache.
`void sys_flush_caches( void );`	First call `sys_dcache_flush_all`(). Then call `sys_icache_invalidate_all`().
`void sys_flush_cache_line( void *addr );`	First `addr` is word-aligned. Then call `sys_dcache_flush_addr`(). Then call `sys_icache_invalidate_addr`().

5.3 Other Cache Issues

A potential source of errors is the access of two memory locations contained within the same cache line, both cached and uncached. If, for example, an address is written uncached, it may be overwritten later by hardware (in case of writeback caches) if the corresponding cache line was valid at the time and later evicted.

Another potential source of errors is the loading of an application to memory. Since this is done by writing the instructions to memory using `store word` instructions (D-cache domain), it is important that the I-cache is invalidated, so that it will be refilled before executing new instructions. To make sure the I-cache refill is performed on the correct data, the D-cache must be flushed to physical memory after loading the application. Also, if the application executes uncached, it is important to flush the D-cache before starting to load the application. All of this is handled by the `load` and `gdb` commands.

I-cache invalidation is followed by a call to `sys_flush_pipeline`(), which performs an `eret` in order to flush the CPU pipeline. This is necessary because instructions in the cache line(s) being invalidated may already be in the CPU pipeline.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

The `copy` and `disk` commands also flush the caches before and after copying data. This is done because these commands are expected to be frequently used for moving applications between, for example, Flash and RAM, and this requires flushing the D-cache and invalidating the I-cache, as previously described. If the user does not want the command to flush the caches, he may use the `-f` option.

Otherwise, YAMON does not usually flush the caches “behind the back” of the user. So, if the user issues an “edit” command to uncached memory, and a memory location within the same cache line has previously been accessed cached, it is the responsibility of the user to make sure the D-cache has been flushed by issuing the `flush -d` command.

The following commands are the only commands which take care of flushing the caches:

- `load`
- `gdb`
- `copy`
- `disk`
- `scpu` (see Chapter 13, “CPU Reconfiguration” on page 65)
- `cache`

YAMON itself is compiled for KSEG0 (by default cached) addresses. Data is generally accessed cached, except in the case of drivers accessing DMA buffers or memory-mapped hardware devices. For systems with coherent IO DMA data buffers are alloacted in cached space to ensure the data is accessed with consistent cache coherency attributes.

If KSEG0 has been configured to be uncached (using the `cache` command), cache flushing will be disabled, so in this case TLB-mapped, cached applications should not be loaded.

5.4 TLB

YAMON performs no TLB operations. The TLB is not initialised nor is it managed.

However, YAMON includes the command `tlb`, which is used to initialise and setup the TLB.

Also, YAMON commands will validate addresses and issue error messages if, for example, an address in a mapped memory range (e.g., KUSEG) is attempted without proper TLB setup.

Address validation is performed by the function `sys_validate_range`() in the file:

```
sys/sys.c
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 6

Shell

This chapter provides an overview of the shell and the shell commands that are supported by YAMON. For a detailed description of shell commands, refer to [1].

6.1 Introduction

The shell parses commands typed on the command line and includes command line history and editing.

The main shell function is `shell`() in the file:

```
shell/shell.c
```

The shell is started by the function `shell_setup`() in the file:

```
shell/shell_init.c
```

`shell_setup`() installs functions accessible from applications through the mechanism described in Section 8.3.1 "Application API".

`shell_setup`() then calls `shell_arch`(), which calls initialisation functions for each shell command supported for a specific board/CPU, and registers the commands using the mechanism described in Section 6.2 "Shell Commands". `shell_setup`() then starts the shell, by calling the `shell`() function.

`shell`() first does some initialisation including registration of the shell error messages. `shell`() then stores the current context so that the shell may later be reentered by a call to `shell_reenter`(). The shell is reentered after an NMI interrupt, a cache error exception, or any other exception occurring in YAMON and causing a context dump (for example, triggered by a '`port -a 1`' command).

`shell`() then enters an infinite loop (implemented in function `command_loop`()), performing the following actions:

- Print prompt.
- Receive command line.
- Remove excess spaces.
- Add line to command stack.
- Call function `execute_line`().

The function `execute_line`() performs the following actions:

- Split line in "`;`" separated sub commands.
- Determine repeat count (indicated by +<*n*> at the beginning of a command line) for line, and repeat the following actions the requested number of times.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

- For each sub command, perform the following actions.
  - Expand environment variables.
  - Call `execute_line`() recursively, so that further "`;`" separation (caused by expansion of environment variables) may occur. This allows for "alias" like use of environment variables (like `setenv x "echo abc; echo def"; $x`). In the recursive call, instead of expanding environment variables, the sub command is divided into tokens and executed by calling function `execute_command`(). The return code from `execute_command`() is used for error handling by calling function `shell_command_error`().

Note that environment variables are not expanded recursively, in order to avoid problems with cyclic dependencies. This is illustrated in the following example:

```
YAMON> setenv x 'echo $y'
YAMON> setenv y 'echo $x'
YAMON> $x
$y
YAMON>
```

The function `execute_command`() performs the following actions:

- Lookup function registered for the command name.
- Call function using traditional `argc`, `argv` based argument passing.

As an example, assume the user has created an environment variable 'test' using the following command:

```
setenv test Hello
```

Now, assume the user enters the following line:

```
echo $test "world !"
```

This will be expanded to three tokens, which are passed to the function registered for the "`echo`" command:

```
echo
Hello
world !
```

The shell implements command line history and editing. Previous commands may be recalled by typing Ctrl-p or arrow-up.

The shell accepts the control codes shown in Table 6.1 .

VT-100 control sequences are used for the arrows ("ESC[A", "ESC[B", "ESC[C", "ESC[D").

Table 6.1 Command Line Recall/Editing Commands

Name	Description
Ctrl-p / arrow-up	Recall previous command in command stack (do not perform it).
Ctrl-n / arrow-down	Recall next command in command stack.
Ctrl-a	Move to first character.
Ctrl-e	Move to last character.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 6.1 Command Line Recall/Editing Commands (Continued)

Name	Description
Ctrl-b / arrow-left	Move one character left.
Ctrl-f / arrow-right	Move one character right.
Ctrl-d	Delete character at cursor position.
Ctrl-h / DEL	Delete character to the left of cursor position.
Ctrl-k	Delete characters from cursor position to end of line.
Ctrl-u	Delete line.
Ctrl-c	Cancel current line.
TAB	Command completion.

Commands may be auto completed by pressing TAB. Also, the shell attempts to auto complete commands when parsing them. However, a minimum 2 characters must be typed before auto completing. For example, if the user enters "`he`", the command "`help`" will be performed.

6.2 Shell Commands

The following subsections describe aspects of shell commands.

6.2.1 Command Registration

Shell commands are defined in the file:

```
include/shell_api.h.
```

`shell_api`.h defines the following command structure:

```
typedef struct
{
    char    *name;  /* Name of command */
    t_func  func;   /* Function implementing cmd */
    char    *syntax;/* Syntax of command */
    char    *descr; /* Detailed description of cmd */
    t_cmd_option*options;/* Command options */
    UINT32  option_count;/* Number of options */
    bool    secret; /* if TRUE, help will ignore */
}
t_cmd;
```

The `t_cmd` structure contains the information required to define a command. Table 6.2 describes the fields of the structure.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 6.2 t_cmd Structure

Field	Description
name	Null terminated string containing the name of the command.
func	The function implementing the command.
syntax	Null terminated string defining the syntax of the command. Used only by the 'help' command and for displaying the syntax in case of syntax errors.
descr	Null terminated string containing a textual description of the command. Used only by the 'help' command.
options	Array of options defined using the t_cmd_option structure. Used only by the 'help' command and possibly the command itself.
option_count	Number of options. Used only by the 'help' command and possibly the command itself.
secret	Boolean variable specifying whether the command should be listed when issuing a '**help**' command. 'help -a' will ignore this variable and list all commands. This variable should usually be set to FALSE.

The t_cmd_option structure is defined in shell_api.h and has the following layout:

```
typedef struct
{
    char  *option;/* Name of option */
    char  *descr;/* Description of option */
}
t_cmd_option;
```

The fields of the t_cmd_option structure are described in Table 6.3.

Table 6.3 t_cmd_option Structure

Field	Description
option	Null terminated string containing the name of the option. All options will be prefixed with the character '-'.
descr	Null terminated string containing a textual description of the option. Used by the 'help' command.

A command is registered by calling the following function also defined in shell_api.h:

```
void
shell_register_cmd(
    t_cmd *cmd )/* Command to be registered */
```

The function implementing a command has the following format:

```
UINT32
name(
    UINT32 argc,
    char **argv )
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

The maximum number of commands registered is defined by the symbol `MAX_COMMANDS` (50) in shell/`shell_init.c`.

6.2.2 Command Invocation

Repeating the above example, assume the user has created an environment variable '`test`' using the following command:

```
setenv test Hello
```

Now, assume the user enters the following line:

```
echo $test "world !"
```

The shell will pass the following arguments to the function registered with name '`echo`':

```
argc = 3
argv[0] = "echo"
argv[1] = "Hello"
argv[2] = "world !"
```

The shell will change line (print '\n') before passing control to a command. The shell will not change line before printing the new prompt upon command return.

6.2.3 Error Reporting

The command function returns an error code as described in Chapter 4, "Error Handling" on page 25. The following generic code indicates "no error":

```
OK
```

A command may request additional information to be displayed by setting the string pointer '`shell_error_data`'. This should always be done in case of illegal options, in which case '`shell_error_data`' should point to the string containing the illegal option.

A command may also include a hint for solving a specific error by setting the string pointer '`shell_error_hint`'.

The shell looks up the error text using SYSCON as described in Chapter 4, "Error Handling" on page 25 and displays it.

6.2.4 Command Input/Output

The following subsections describe how shell commands may receive data from the UART and transmit data to the UART.

6.2.4.1 Input

Commands may receive characters from the terminal by calling the following functions implemented in `sys/sys.c` and defined in `include/sys_api.h`:

```
bool
sys_getchar(
    UINT32 port,
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
    char *ch );

bool
sys_getchar_ctrlc(
    UINT32 port );

#define GETCHAR(port, ch)sys_getchar(port, ch)
#define GETCHAR_CTRLC(port)sys_getchar_ctrlc(port)
```

These function take as a parameter the port to be used (tty0 or tty1). The port values are defined in `include/sysdefs.h`:

```
#define PORT_TTY0    0
#define PORT_TTY1    1
#define PORT_NET     2
```

`PORT_NET` corresponds to the TFTP/UDP/Ethernet connection used by the `load` command (see Section 8.1 “Loading Applications”).

`sys_getchar`() polls for a received character. In case a character was received, it is written to the char pointed to by the ’ch’ parameter, and the function returns TRUE. If no character is available, it returns FALSE.

`sys_getchar_ctrlc`() does not actually read a character, but is used for determining whether a ctrl-c has been pressed. It returns TRUE if ctrl-c has been detected, otherwise FALSE.

The functions do not echo received characters.

6.2.4.2 Output

Commands may output data to the terminal in two ways:

- Using the standard library (e.g., `printf`()).
- Using the functions `sys_puts`() or `sys_putchar`().
- Using the functions `shell_puts`() or `shell_putc`().

`printf`() outputs data on tty0 and has the traditional functionality with a format string and a (variable) number of arguments.

`sys_puts`() and `sys_putchar`() are implemented in `sys/sys.c` and defined in `sys_api.h`. They output data on tty0 or tty1 (as defined by `PORT_TTY0` and `PORT_TTY1` in the subsection "Input" above):

```
void
sys_puts(
    UINT32 port,
    char *s )
void
sys_putchar(
    UINT32 port,
    char   ch )

#define PUTS(port, s)sys_puts(port, s)
#define PUTCHAR(port, ch )sys_putchar(port, ch)
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

shell_puts() and shell_putc() are implemented in shell/shell.c and defined in include/shell_api.h:

```
bool
shell_puts(
    char   *string,
    UINT32 indent )

bool
shell_putc(
    char   ch,
    UINT32 indent )

#define SHELL_PUTS( s )shell_puts( s, 0 )
#define SHELL_PUTS_INDENT( s, indent )shell_puts( s, indent )
#define SHELL_PUTC( c )shell_putc( c, 0 )
#define SHELL_PUTC_INDENT( c, indent )shell_putc( c, indent )
```

shell_puts() and shell_putc() perform the following actions (sys_puts() and sys_putchar() do not do this):

- Always output to tty0.
- Keep track of the number of characters displayed on the current line.
- Allow request for a specific indentation.
- Keep track of the current line number.
- Each MON_DEF_LINEMAX (24) numbers of lines, display the following message and await a keypress before replacing the message with the requested text and continuing.

  ```
  Press any key (Ctrl-C to break, Enter to singlestep)
  ```

- In case a ctrl-c is detected (at any line), indicate this to the calling function by returning TRUE. Otherwise, return FALSE.

The “Press any key” message feature may be disabled/enabled by calling the following function (implemented in shell.c and defined in shell_api.h):

```
void
shell_setmore(
    bool enable_more );

#define SHELL_DISABLE_MORE    shell_setmore( FALSE )
#define SHELL_ENABLE_MORE     shell_setmore( TRUE )
```

Note that the printf() function will not update the line number. It is recommended that shell commands use either printf()/sys_puts()/sys_putchar() or shell_puts()/shell_putc(), not both.

Please observe that the ctrl-c information is volatile and disappears after sys_getchar(), sys_getchar_ctrlc(), shell_puts() and shell_putc(). In order to catch ctrl-c correctly, the return value of these functions must always be checked.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 7

Exception Handling (EXCEP Module)

This chapter describes how exceptions are handled by YAMON.

7.1 Overview

Exceptions are handled in four steps:

1. Assembler code is installed at exception vector locations storing context and forwarding vectorized exceptions to a single "C" language entry point for non-EJTAG exceptions (`exception_sr`()) and a "C" language entry point for EJTAG exceptions (`exception_ejtag`()).

2. First level branch to Exception Service Routines "ESR" registered for the particular exception cause code. Special handling of NMI, cache error, and EJTAG exceptions.

3. The function "`interrupt_sr`()" is registered as the ESR for cause code "Interrupt". This function performs a second level branch to CPU Interrupt Service Routines "CPU ISR" for the particular sw and hw interrupt number.

4. A special "CPU ISR" (`controller_sr`()) may be registered for the hw interrupt used by the interrupt controller (if one is used). This function performs a third level branch to Interrupt Controller Interrupt Service Routines "IC ISR" for the particular interrupt line.

Each level may be intercepted by application programs by hooking of the vector or registration of a specific or general entry.

The following exceptions are handled:

- Reset exception (see Section 7.2 “Reset Exception”)
- NMI exception (see Section 7.3 “NMI Exception”)
- EJTAG exception (see Section 7.4 “EJTAG Exception”)
- CACHE ERROR exception (see Section 7.5 “Cache Error Exception”)
- FPU exception (see Section 7.6 “FPU Exception”)
- Other exceptions, including interrupts (see Section 7.7 “Exception Handlers”)

The following is a summary of RAM-based exception vectors, including two vectors allocated by YAMON:

- 0x80000000 (TLB refill)
- 0x80000080 (XTLB refill)

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

- 0xa0000100 (Cache error)
- 0x80000180 (General exception)
- 0x80000200 (Used for Interrupts since "*IV*" field of *CAUSE* register is set)
- 0x80000300 (Allocated by YAMON, used for legacy EJTAG exceptions)
- 0x80000380 (Allocated by YAMON, used for legacy NMI exceptions)
- 0x80000a00 (Allocated by YAMON, used for EJTAG exceptions)
- 0x80000a80 (Allocated by YAMON, used for NMI exceptions

The last two entries allow applications to receive EJTAG and NMI exceptions by installing code at those vector addresses. Prior to YAMON 2.07, the legacy locations were used for the EJTAG & NMI vectors, but these clashed with the EIC interrupt vector table. For backwards compatibility, YAMON installs a jump to the legacy exception vectors when EIC mode is not being used.

Exception handling is performed by the EXCEP module located in the following two directories:

- `exception`: Architecture-independent code.
- `arch/exception`: Architecture-dependent code.

The EXCEP module initialisation `EXCEP_init()` code will call `EXCEP_install_exc_in_ram()` to setup the various exception vector locations and clear the *BEV* bit of the CP0 *STATUS* register.

Before the exception vector locations have been set up by the EXCEP initialisation code, the *BEV* bit of the *STATUS* register will be set and interrupts will be disabled. Exceptions will cause exception handlers located in flash memory (0xbfc00xx0) to be invoked. Except for a new Reset exception, an NMI, or an EJTAG exception, these handlers will simply halt YAMON (loop forever as coded in file `arch/reset/bootvector/reset.S`). In the case of a cache error, the message "CacheEr" will be displayed in the LED display (if the board supports this).

7.2 Reset Exception

A Reset exception will cause the YAMON code located at the reset vector location (0xbfc00000) to be invoked. This is described in Chapter 11, "Initialization" on page 59.

7.3 NMI Exception

An NMI exception will also cause the YAMON code located at the reset vector location (0xbfc00000) to be invoked. After some early initialisation code (see Chapter 11, "Initialization" on page 59), YAMON will determine in a platform-specific way whether it was an NMI exception that caused the jump to location 0xbfc00000. If so, the text "NMI" will be displayed in the ASCII-display (if the platform supports this) and a jump is performed to address 0x80000a80.

The EXCEP module will install code at 0x80000a80 that performs the following:

- Write the exception vector offset (0xa80) to a system variable to indicate this is an NMI exception.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

- Jump to the function `exc_handler`(), which is the function handling most other exceptions as well.

For further actions, see Section 7.7.1 “exc_handler()”.

7.4 EJTAG Exception

An EJTAG exception will cause instructions to be fetched from address 0xbfc00480 (unless an EJTAG probe has requested instructions to be fetched from the probe).

The YAMON code at 0xbfc00480 will jump to the address 0x80000a00 with unmodified registers.

The EXCEP module will install code at 0x80000a00 that performs the following:

- Save register k1 in a reserved CP0 register (CP0 *DESAVE*).
- Save register context including k0 and k1 at a specific EJTAG context structure.
- Set up stack pointer to a specific EJTAG exception stack.
- Jump to the function `exception_ejtag`(), where a branch is taken to an "Exception Service Routine" (ESR) specific to the EJTAG cause code (or perform a context dump and reenter shell if no ESR is registered).

7.5 Cache Error Exception

The EXCEP module will install code at the cache error exception vector (0xa0000100) performing the following:

- Jump to function `exc_handler_cacheerr`(), which performs the following:
  - Read CP0 *CONFIG* register and store it in a system variable, so that the value may later be written by a context dump.
  - Change *K0* field of CP0 *CONFIG* so that KSEG0 now executes uncached.
  - Jump to the function `exc_handler`(), which is the function handling most other exceptions as well.

For further actions, see Section 7.7.1 “exc_handler()”.

7.6 FPU Exception

An FPU exception will normally follow default action, i.e., jump to `exc_handler`() (see Section 7.7.1 “exc_handler()”. It is however possible to enable a built-in FPU emulator. See Chapter 9, “FPU Emulator” on page 53 and [1] for details.

7.7 Exception Handlers

7.7.1 exc_handler()

Except for Reset and EJTAG exceptions, all exceptions eventually reach `exc_handler`(), assuming *BEV* field of CP0 *STATUS* register is 0.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

exc_handler() will perform the following:

- Store context (CPU, CP0, and possibly FPU registers). In case of a MIPS32/MIPS64 Release 2 CPU supporting register shadow sets, save CPU registers corresponding to the shadow set that was in use when the exception was taken are stored.
- Set stack pointer (sp) to an exception stack.
- Jump to the function exception_sr() with the cause code field of CP0 *CAUSE* register as argument. The exception vector offset is also passed as an argument to distinguish between NMI and Cache error exceptions.

Note that a similar function exc_handler_ejtag() is used for EJTAG exceptions.

7.7.2 exception_sr() branch out

exception_sr() is the “first-level branch out” exception handler, working in parallel with exception_ejtag():

First level:

NMI will cause "super default" action, which will print a register dump to tty0 and restart YAMON shell.

A branch is taken to an "Exception Service Routine" (ESR) specific to each cause code.

An "Exception Service Routine" (ESR) may be registered for any specific exception cause code. A default ESR may be registered for handling exceptions not handled by any specific ESR. If no ESR (not even a default ESR) has been registered, a “super default” handler is invoked, which will print a register dump to tty0 and restart YAMON shell.

Registration of an ESR is done using the EXCEP module API as described in Section 7.8 “EXCEP API”.

Second level:

A specific ESR is the function interrupt_sr(), which is registered by YAMON for the cause code "Interrupt".

In interrupt_sr() a branch is taken to an "Interrupt Service Routine" (CPU_ISR) specific to each hardware or software interrupt line of the CPU.

An "Interrupt Service Routine” (CPU_ISR) may be registered for any specific hardware or software interrupt line of the CPU. A default CPU_ISR may be registered for handling hw/sw interrupts not handled by any other CPU_ISR. interrupt_sr() will pass control to the CPU_ISR registered for the interrupt that occurred (or the default CPU_ISR). If no CPU_ISR (not even a default CPU_ISR) has been registered, the "super default" handler is invoked as described above.

Registration of an CPU_ISR is done using the EXCEP module API as described in Section 7.8 “EXCEP API”.

Third level:

A specific CPU_ISR is the function controller_sr(), which is registered by YAMON for the interrupt port used by the interrupt controller (if an interrupt controller is available).

In controller_sr() a branch is taken to an "Interrupt Controller Routine" (IC_ISR) specific to each line of the interrupt controller.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

An interrupt controller IC_ISR may be registered for any specific line of the interrupt controller. A default IC_ISR may be registered for handling interrupts on lines not handled by any other IC_ISR. The function controller_sr() will pass control to the IC_ISR registered for the interrupt that occurred (or the default IC_ISR). If no IC_ISR (not even a default IC_ISR) has been registered, the "super default" handler is invoked as described above.

Registration of an IC_ISR is done using the EXCEP module API as described in Section 7.8 “EXCEP API”.

If an ESR returns, a jump is performed to function EXCEP_exc_handler_ret(), which will restore the context and perform an eret or deret instruction based on whether the exception was “normal” or “ejtag”.

The ESR registered for interrupts (interrupt_sr()) will return this way unless the super default handler is called (interrupt with no registered ISR), in which case YAMON will print a register dump and restart.

Instead of returning, an ESR may itself call EXCEP_exc_handler_ret(). EXCEP_exc_handler_ret() has the following format:

```
/************************************************************************
 *
 *                          EXCEP_exc_handler_ret
 *  Description :
 *  -------------
 *  Restore context and return from exception
 *
 *  Return values :
 *  ---------------
 *  None
 ************************************************************************/
void
EXCEP_exc_handler_ret(
 t_gdb_regs *context );
```

The “context” parameter contains a pointer to a structure containing the context (CPU, CP0 and FPU register values) to be restored.

By calling EXCEP_exc_handler_ret() directly, a registered ESR may control the context to be restored.

Another function EXCEP_exc_handler_ret_ss() is used for toggling between YAMON and the application context, as described in Chapter 8, “Applications” on page 49. This function first makes sure that CPU register shadow set 0 is used (for MIPS32/MIPS64 Release 2 CPUs including shadow sets) since the application might have changed this. It then calls EXCEP_exc_handler_ret().

7.8 EXCEP API

The EXCEP module has the following API functions (see excep_api.h for details):

```
EXCEP_init()
EXCEP_register_esr()
EXCEP_deregister_esr()
EXCEP_register_cpu_isr()
EXCEP_deregister_cpu_isr()
EXCEP_register_ic_isr()
EXCEP_deregister_ic_isr()
EXCEP_store_handlers()
EXCEP_set_handlers()
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
EXCEP_install_exc_in_ram()
EXCEP_save_context()
EXCEP_get_context_ptr()
EXCEP_exc_handler_ret()
EXCEP_exc_handler_ret_ss()
EXCEP_print_context()
EXCEP_run_default_esr_handler()
```

Only one ESR may be registered for a specific exception. On the other hand, more than one `CPU_ISR` or `IC_ISR` may be registered for a specific interrupt.

When registering an ESR, it may be requested that the ESR is called in “raw” mode, i.e., with context (CPU, CP0 and FPU registers) in the exact state it was when the exception occurred, except for k0 and k1 (in case of an EJTAG exception, the exact state includes k0 and k1).

7.9 Platform Adaptation of Interrupt Controller

In order to adapt YAMON for a new interrupt controller, modifications must be made to the file:

```
arch/exception/platform/excep_platform.c
```

The file contains the following functions that must be modified in order to support the new platform:

- `arch_excep_init_intctrl`()

  This function initialises the interrupt controller (if any). The function must set the two output variables ’`ic_count`’ and ’`ic_int`’.

- `arch_excep_enable_int`()

- `arch_excep_disable_int`()

  These functions enable/disable a specific line of the interrupt controller.

- `arch_excep_pending`()

  This function returns the interrupt status value. One bit must be set for each pending interrupt. It is also allowed to just set the bit for the highest priority pending interrupt.

- `arch_excep_eoi`()

  On some platforms, an "End Of Interrupt" cycle must be performed. This is the case for the Southbridge controller on the Malta board, but is not the case for the interrupt controller on the Atlas board.

Besides adapting `excep_platform.c`, device drivers must configure devices to generate interrupts and register ISRs (Interrupt Service Routines).

For some platforms (e.g., Malta), some interrupt related setup may also be performed in the file:

```
arch/init/platform/init_platform.c.
```

For example, in case of Malta, the IRQ numbers of the units of the SMSC Super I/O controller (UARTs, parallel port, etc.) are configured in this file.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 8

Applications

This chapter describes how applicatons are loaded and executed using YAMON.

8.1 Loading Applications

Applications are loaded using the `load` command and executed using the `go` command or, under debugger control, using the `gdb` command. The `load` command implemented in:

```
shell/load.c
```

The shell `load` command invokes the function `loader_image`() in file `load/loader.c`, which controls the loading and interprets the file formats.

Currently, only Motorola S-record files as defined in Ref [1] are supported.

`loader_image`() loads the S-records from one of the two UARTS or from Ethernet using the TFTP/UDP protocol (on platforms supporting Ethernet).

8.2 Contexts

YAMON operates with four different contexts:

- YAMON context - saved in structure `shell_context` while a `go` or `gdb` command is executing, with stack space allocated in `init.S`.
- Exception context - with stack space allocated in `excep_s.S`. This context includes set up of register gp to YAMON's value, which enables exception routines to access all of YAMON's variables and functions. At entry, current context is saved in structure `exc_context`.
- EJTAG exception context- with stack space allocated in `excep_s.S`. This context includes set up of register gp to YAMON's value, which enables exception routines to access all of YAMON's variables and functions. At entry, the current context is saved in the structure `ejtag_context`.
- Application context - saved in structure `appl_context`, with stack space allocated in `init.S`.

8.3 Go Command

The `go` command first examines the arguments and sets up the initial application context. This is defined by the fields of the structure `user_context` of type `t_gdb_regs` as described in Table 8.1. Then, the application is invoked using the following mechanism - which is by the way shared with the command "`gdb`":

First, the application context (`appl_context`) is set up as shown in Table 8.1.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Then shell_to_shift_user() (in shell/go.c) is called, which causes a shift to the application context in the following way:

- The function appl_exception() is registered as default ESR (Exception Service Routine). If the application is being run through GDB, appl_exception() is also registered for the BREAK exception.
- The current context (YAMON context) is saved so that it may be restored later.
- EXCEP_exc_handler_ret() (see Chapter 7, “Exception Handling (EXCEP Module)” on page 41) is called with a pointer to the application context. EXCEP_exc_handler_ret() will load that context and issue an ERET, thus causing code to continue at the location pointed to by its EPC, which is the application’s entry point.

Applications may return to YAMON by the method described in Section 8.3.1 “Application API”.

Table 8.1 Initial Application Context

Field	Value
reg4 (a0)	Set to the argument count.
reg5 (a1)	Pointer to array of strings holding the arguments. argv[0] == “go” in case application is started by “go” command. argv[0] == “gdb” in case application is started by “gdb” command.
reg6 (a2)	Pointer to table holding environment variables.
reg7 (a3)	Size of memory (in bytes).
reg29 (sp)	4 words below top of memory range reserved for user stack (size defined by symbol SYS_APPL_STACK_SIZE defined in include/sys_api.h).
reg31 (ra)	Return address. Application may jump to this address in order to exit and return to YAMON.
Other CPU registers	0
FPU registers (if available)	0
cp0_status	Same as YAMON context except that IE bit is cleared thus disabling interrupts.
cp0_epc	Entry point of application obtained from “load” command or as a parameter to “go”.
Other CP0 registers	Identical to YAMON context.
FPU control registers (if available)	Identical to YAMON context.

8.3.1 Application API

YAMON holds a table at flash memory location 0x1fc00500 holding addresses of functions that applications may invoke in order to shift back to YAMON context and perform specific actions.

One of these functions is the exit function. Calling the exit function will cause the application to end. The same result may be obtained by jumping to the address stored in register $31 (ra) when the application was invoked. Register ra holds the address of the function shell_return() in file shell/appl_if.S. The only difference between the exit function and shell_return() is that the exit function transfers its return value in register a0, while shell_return() uses register v0.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

The application API is described in Ref [1]. This section describes how this interface is implemented. The application API allows applications to request YAMON to perform the following operations (documented in the header file `include/yamon_api.h`):

- Print string to tty0 (either zero terminated or using a character count).
- Get character from tty0.
- Flush caches.
- Register/deregister Exception Service Routines (ESRs) and Interrupt Service Routines (ISRs). ESRs are always called in “raw” mode, i.e., with context in the exact state it was when the exception occurred (except for k0 and k1 registers). Note that the YAMON GDB stub must “own” the BREAK exception.
- Access SYSCON module (read object values).
- Exit application.

The function pointers are stored in a table at base address 0x1fc00500. The function pointed to is found in the file `appl_if.S`. That function (and the similar function `shell_return`() to which `ra` points when application is started) will cause the YAMON context to be entered by the following mechanism:

- In assembler level, Save CP0 *STATUS* register and disable interrupts.
- Save all general purpose registers to `appl_context`.
- Set stack pointer (`sp`) to the current YAMON `sp` value, so that the active part of the YAMON stack is untouched.
- Call `appl_shell_func`() to perform the requested function in c-level.
- If the function requested is “`exit`”, or the application simply jumped to the address contained in register ra when application was started, `EXCEP_exc_handler_ret_ss`() (see Chapter 7, “Exception Handling (EXCEP Module)” on page 41) is called with a pointer to the previously saved YAMON context, returning back into `shell_shift_to_user`() to end the YAMON "go" comand.
- Otherwise `appl_shell_func`() will update `appl_context` register `v0` with its return value and simply return to assembler level.
- At the assembler level, restore all general purpose registers and CP0 *STATUS* register.
- Return to the application.

8.3.2 GDB Command

This section describes the GDB interface implemented by the shell command “`gdb`”. The supported subset of the GDB “Standard Remote Protocol” (with extensions for Algorithmics SDE-GDB) is described in Ref [1].

The "`gdb`" shell command is implemented in the file:

```
shell/gdb.c
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

The gdb command function gdb() first sets up the initial application context in the same way as the "go" command (see Table 8.1).

GDB may request the target (i.e., YAMON) to single-step code, in which case YAMON replaces the next instruction of the application with a BREAK instruction (and storing the old instruction, so that it may be rewritten later). Actually, in case of branches, two BREAK instructions are inserted, one at the instruction following the branch delay slot, and one at the target address of the branch.

Setting breakpoints in exception-handling code is not supported.

When the GDB debugger issues a command requesting an application to be executed, like the 'C' (Continue) or 'S' (Singlestep) command, the function shell_shift_to_user() is called, as described in Section 8.2 "Contexts", with a pointer to the desired user context (with cp0_epc set to the start address). shell_shift_to_user() is notified that this application is being debugged, so that exceptions will not cause the context to be displayed and YAMON halted, but rather will cause shell_shift_to_user() to return to gdb(), so that GDB may be notified of the exception (with special handling of BREAK exception) and act accordingly.

When the function shell_shift_to_user() returns, it will have updated the application context structure with the new context (including new EPC). The gdb() function determines if single-stepping was set up, and in this case restores the original instruction(s). Then it responds to GDB and starts polling GDB again for the next command on tty1.

Two options are available for the gdb command. They are:

- -v (verbose). This option will cause all requests from GDB and all responses from YAMON to be displayed on tty0.
- -c (checksum off). This option will disable the checksum validation used in GDB commands. This is very useful for debugging the GDB protocol, because it makes it convenient to enter commands manually (using a terminal rather than the GDB host connected to tty1) without having to calculate checksums. The user may simply terminate commands with the sequence #00 (or any other dummy checksum).

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 9

FPU Emulator

This chapter describes YAMON's emulation of floating-point operations.

9.1 Overview

Floating-point operations on very small numbers (denormalized numbers) and illegal numbers (NaNs) must be handled by software. Also, some IEEE exceptions and exceptional results will cause an FPU exception. The FPU emulator is thus mandatory for full IEEE compliance, even when the system has a hardware FPU.

The FPU exception handler is a GPL licensed stand-alone module, i.e., it comes with a complete exception handler. YAMON will call it without stacking any registers. If the emulation succeeds, the emulator will perform a return from exception. Otherwise, the YAMON exception handler will be called.

The exception handler is also optional: not including it in a YAMON build will result in a system that does not follow IEEE floating-pointfloating-point behaviour accurately (with or without a hardware FPU), but it is fully functional and allows modifications to YAMON to have more flexible licensing options.

9.2 Support

The emulator supports:

- MIPS32 FPU instructions
- MIPS64 FPU instructions
- 32-bit addressing

The emulator does NOT support:

- MIPS-3D ASE FPU instructions
- Paried Single FPU instructions
- 64-bit addressing

9.3 Directory Structure

The directory structure is shown in Table 9.1.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Table 9.1 FPU Emulation Directories

Directory	Contents
`fpuemul`	Main directory. Makefile, handler.S etc.
`fpuemul/include/`	Header files
`fpuemul/include/asm`	Header files
`fpuemul/include/linux`	Header files
`fpuemul/include/sys`	Header files
`fpuemul/math`	IEEE754 functions and header files

All the files in `fpuemul/math` correspond to those in the MIPS Linux distribution.

9.4 Cause Codes

There are two relevant exception cause codes (in the CP0 *CAUSE* register) for the FPU emulator:

- 15, FPE, floating-point exception. This exception occurs when the hardware FPU operates on numbers it can't handle. When the emulator is enabled for this exception, it ensures that denormalized numbers and large numbers in conversions are handled correctly.

- 11, CpU, coprocessor unusable. This exception occurs whenever a COP1 instruction is encountered and there is no FPU, or the FPU has been disabled in the CP0 *STATUS* register. When the emulator is enabled for this exception, it is possible to emulate all FPU instructions. This is not recommended for performance-critical applications; a software floating-point library should be used instead.

9.5 32/64-bit Stacking

The exception handler will automatically determine whether all registers must be saved as 32-bit registers or 64-bit registers. When YAMON is initialized it analyzes the CPU. If the CPU can run in 64-bit mode, the variable `sys_64bit` is set to 1. The FPU emulator uses the external variable `sys_64bit` to determine whether to stack all registers as 32-bit or 64-bit registers.

9.6 API

The FPU emulator offers a single function:

- `FPUEMUL_handler`

The handler takes a single argument in `k1`., which is expected to hold the address of the default handler. If the emulation succeeds, an `eret` will cause the emulator to return directly from the exception. If the emulation fails (illegal instruction or similar), the emulator will jump to the handler pointed to by `k1`.

The global structure `FPUEMUL_stat` contains statistics for the emulator. See `include/asm/fpu_emulator.h` for details. The struct has a string "`error_text`" which is set if a problem occurred with the emulation. YAMON is expected to display this text on failure.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

9.7 Trampoline Code

A special problem exists when emulating FP branch instructions. **This is only relevant if there is no H/W FPU, or it is disabled via the CU1 bit in SR**. When the FP instruction has been emulated, the instruction in the branch delay must also be emulated. To avoid having a complete integer instruction emulator, a so-called *trampoline* is used. The instruction from the branch delay is copied to the area above the user's stack pointer, followed by an invalid COP1 instruction. The EPC is set to point the this area before calling `eret`. The invalid COP1 instruction ensures that the emulator will regain control and in turn setup the correct EPC produced by the branch. Note that this is self-modifying code, so the caches must be invalidated.

WARNING

If the emulator is ported to a system which supports multiple users (e.g., Linux), the trampoline code must be thoroughly inspected. Beware not to execute the outstanding branch delay instruction from kernel space. The instruction must be copied to and executed from user space. A suggested implementation is as follows:

- Make sure all users have 16 (20 for 64-bit addressing) bytes of stack allocated for this purpose.
- Copy the outstanding branch delay instruction to the user's stack (SP+0).
- Place an invalid COP1 instruction after it (SP+4).
- Place a cookie at (SP+8).
- Save the next EPC (SP+12).
- Execute `eret`.
- In the exception handler, check that the instruction causing the exception was the predefined invalid COP1 instruction. Also check the cookie, and then restore the EPC (from SP+12).
- Execute `eret`.

9.8 How to Extract the Emulator

- Make sure that all directories and sub-directories in `fpuemul` are available.
- Fix the trampoline code as necessary.
- Check the file `porting.S` for the method of addressing user space.
- Make available the 8-bit variable `sys_64bit` (or change the code). 0 indicates that we are running on a 32-bit system, 1 indicates that we are running on a 64-bit system.
- Let the general exception handler call `FPUEMUL_handler` for FP exceptions. Make sure that `k1` points to the default exception handler.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

9.9 GPL-free YAMON build

In its basic form, YAMON has a soft FPU and is subject to the GPL. This means that regardless of any hardware FPU support, the system is fully compliant with IEEE floating-point operations. YAMON itself uses no FPU operations, and will function normally with no FPU and no software support.

If a GPL-free YAMON build is required, the soft FPU can be removed from the build by running a helper shell script from within the bin directory:

```
cd bin
./nogpl.sh
```

There are some issues to note about operation without the soft FPU:

- Any attempt to use instructions that access the FPU will result in the floating-point exception path being followed, and the operation terminated with SIGFPU.
- If the FPU environment variable is set to use the FPU emulator before flashing the system with a non-FPU YAMON, the contents of the environment variable will be erased on first boot.

If you wish to alter the behaviour of the non-FPU system without using the GPL FPU, use the file `cop1.c`, in the `fpuemul` directory, which contains the function that handles the exception branch from `handler.S`.

For more information regarding the need for a soft FPU on a system with a hardware FPU, consult Chapter 7.4 of *See MIPS Run Linux* (Second Edition) by Dominic Sweetman.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 10

System Header Files

This chapter describes the header files used by YAMON.

The main system header file is:

```
include/sysdefs.h
```

`sysdefs.h` includes, among other things, the following:

- Definitions of integer types (UINT32, UINT16 etc.).
- Macros converting to KSEG0, KSEG1, KUSEG etc. addresses.
- Endianness swapping macros.
- The definition of the UART used by the shell (`DEFAULT_PORT` set to `PORT_TTY0`).
- The definition of the UART used by GDB (`DEFAULT_GDB_PORT` set to `PORT_TTY1`). If a board has only a single tty available, `DEFAULT_GDB_PORT` symbol is ignored and `PORT_TTY0` is used for the GDB port.

Refer to the source file for further information.

Another header file (often used by assembler files) is:

```
arch/include/mips.h
```

`mips.h` includes a few workarounds for bugs in early revisions of 4K family of processors.

`mips.h` further includes the following file holding basic MIPS32 and MIPS64 definitions:

```
arch/include/ArchDefs.h
```

`ArchDefs.h` includes definitions of the CP0 registers and the following naming conventions for CPU general purpose registers:

```
#define    zero       $0
#define    AT         $1
#define    v0         $2
#define    v1         $3
#define    a0         $4
#define    a1         $5
#define    a2         $6
#define    a3         $7
#define    t0         $8
#define    t1         $9
#define    t2         $10
```


Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
#define    t3          $11
#define    t4          $12
#define    t5          $13
#define    t6          $14
#define    t7          $15
#define    s0          $16
#define    s1          $17
#define    s2          $18
#define    s3          $19
#define    s4          $20
#define    s5          $21
#define    s6          $22
#define    s7          $23
#define    t8          $24
#define    t9          $25
#define    k0          $26
#define    k1          $27
#define    gp          $28
#define    sp          $29
#define    s8          $30
#define    fp          $30
#define    ra          $31
```

Again, refer to the source file for further information.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 11

Initialization

This chapter describes the initialization code executed following a reset.

Following a reset, hardware fetches instructions starting at the reset exception vector (0xbfc00000).

Code is actually linked to KSEG0 (configured as cached) address space, but until caches have been initialised, code executes in KSEG1 (uncached) address space. This requires addresses to be converted to KSEG1 space before performing jumps.

Initially, code is executed from flash/(E)PROM. At some point, the code is copied from flash to RAM, and YAMON starts executing from RAM space because executing from RAM is faster.

The code is linked the following way, as defined in the linker scripts:

```
bin/link/link.xn
bin/link/link_el.xn
bin/link/link_eb.xn
```

Based at 0x9fc00000:

- `reset.S`

Based at 0x9fc10000 (little endian) or 0x9fc78000 (big endian):

- `init.S`
- `init_platform_s.S`
- `init_core_s.S`
- `atlas_malta_platform.S`
- `gt64120_core.S`
- `bonito64_core.S`
- `msc01_core.S`
- `sead_platform.S`
- `init_cpu_s.S`
- `cache_cpu.S`

Remaining code is linked at base address 0x80005000 (cached, RAM).

The initialisation code is located in the following directories:

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

- `arch/reset/bootvector`: The file `reset.S`
- `arch/reset`: Architecture dependent assembler files.
- `init/`: Architecture independent code in c file format.
- `arch/init` : Architecture dependent code in c file format.

The function __reset_vector() in file `arch/reset/bootvector/reset.S` is invoked following a reset. This function detects the endianness and jumps to the corresponding little or big endian image. The endian-specific code entry point is the function:

```
__reset_handler()
```

in file `arch/reset/init.S`.

`reset_handler`() is linked at addresses 0x9fc10000 (little endian) and 0x9fc78000 (big endian).

The initialisation code is contained in assembler files and C-files. The first code executed is assembler code, all of which is found in the directory `arch/reset`. Files located in that directory have several things in common:

- Their code is linked to flash and executed from flash
- Their file names are listed separately in linker scripts
- They execute on registers alone, without using any RAM variables or any stack space
- They detect and set up the SDRAM properties (size/speed, etc.) and configure the system controller accordingly, enabling the first C-function to be called.

Other functions performed by the assembler code includes detecting NMI, detecting cache sizes, initialising caches, copying code and initialised data from boot flash/eprom to RAM, and initialising stack pointer (sp).

The first C-function called is `c_entry()` in `init/main.c`. This function performs the remaining initialisation before calling the function `shell_setup`(), which starts the shell. Most of the initialisation requested by `c_entry`() is performed by the function `initmodules`() as described in Chapter 3, “Modules” on page 19.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 12

PCI Configuration (PCI Module)

This chapter describes YAMON's support for PCI devices.

12.1 Introduction

The PCI module detects the available PCI devices and verifies that the devices found belong to one of following categories:

- Known devices of the board
- Device located in PCI slot
- Device located behind a PCI bridge

The PCI module then allocates the PCI memory and I/O maps based on the requirements of the devices and configures the devices accordingly. Some of the known devices may have fixed requirements for memory and I/O mapping.

The PCI module has two public functions that may be used for looking up the configuration for a specific device or a specific memory or I/O region (BAR setup) for a device:

```
pci_lookup_device()
pci_lookup_bar()
```

In order to adapt the PCI module to a specific platform, the following files must be adapted as described in the following sections:

```
arch/pci/platform/pci_platform.c
arch/pci/platform/core/pci_core.c
```

12.2 Adaptation of pci_platform.c

The following subsections describe the functions that must be adapted in `pci_platform.c`.

12.2.1 pci_config()

This function reads the available memory and I/O ranges through SYSCON calls (these objects must be adapted to the platform during initialisation of the SYSCON module, see Section 3.2 "System Configuration (SYSCON module)". The function then sets the following platform specific variables, which must be adapted to the platform:

- `known_devs`: Must point to an array of '`t_known_dev`' structures (defined in `arch/include/pci.h`) describing the known devices. In the case of the Atlas and Malta boards, one additional array element has been reserved for the system controller data. This is setup by the function `arch_pci_config_controller`() (see Section 12.3 "Adaptation of pci_core.c").

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

```
typedef struct
{
    UINT16  vendorid;    /* Vendor ID                    */
    UINT16  devid;       /* Device ID                    */
    UINT8   function;    /* Function number              */
    UINT8   intline;     /* Interrupt line used by device */
    char    *vendor;     /* String holding vendor name   */
    char    *device;     /* String holding device name   */
}
t_known_dev;
```

- known_devs_count: Must be set to the size of the known_devs array (number of known devices).
- intline: The interrupt line used for the system controller.
- bar_req: Mapping of PCI memory and I/O ranges is done through the "Base Address Registers" (BARs) of the PCI devices. If some devices have fixed requirements for mapping of memory and I/O space, the bar_req array must be setup accordingly. bar_req consists of elements of type t_pci_bar_req (defined in arch/include/pci.h):

```
typedef struct
{
    UINT16      vendorid;       /* Vendor ID            */
    UINT16      devid;          /* Device ID            */
    UINT8       function;       /* Function number      */
    t_pci_bar   bar;        /* Requirements for BAR */
}
t_pci_bar_req;
```

The structure t_pci_bar is defined in arch/include/pci_api.h:

```
typedef struct pci_bar
{
    struct pci_bar  *next;     /* Reserved (linked list ptr)          */

    UINT8           pos;       /* BAR position                        */
    UINT8           io;        /* 1 -> IO mapped, 0 -> Memory mapped  */
    UINT32          mask;      /* Mask obtained from device           */
    UINT8           prefetch;  /* Prefetch field obtained from device */
    UINT32          start;     /* PCI side start address of range     */
    UINT32          size;      /* Size of range                       */
    bool            fixed;     /* TRUE -> Fixed location BAR          */
}
t_pci_bar;
```

Only the 'pos', 'io', 'prefetch', 'start' and 'size' fields of t_pci_bar must be setup.

- bar_count: Must be set to the number of such requirements for base address registers.

12.2.2 arch_pci_system_slot()

This function determines whether or not the board is located in a Compact PCI system slot.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

12.2.3 arch_pci_slot()

This function determines whether or not a particular "`Device number`" corresponds to a PCI slot. `arch_pci_slot`() must set the '`number`' variable to the number of the PCI slot. '`number`' is only used for display via the "`info pci`" command.

The "`Device number`" depends on which PCI address bit is used for the *IDSEL* pin of the device:

```
Device number 1 maps to ADP11
Device number 2 maps to ADP12
  ...
Device number 21 maps to ADP31
```

12.2.4 arch_pci_slot_intline()

This function determines the interrupt line (for the interrupt controller) used for the PCI slot identified by the PCI device number.

12.2.5 arch_pci_remote_intline()

This function determines the interrupt controller line used for PCI interrupt pin on Compact PCI connector (only available on the Atlas board).

12.3 Adaptation of pci_core.c

The following subsections describe the functions that must be adapted in `pci_core.c`.

12.3.1 arch_pci_config_controller()

This function is called from the `pci_config`() function described above (see Section 12.2 "Adaptation of pci_platform.c"). It configures the system controller (Northbridge) device according to the setup determined by `pci_config`(), and performs any other PCI-related initialisation of the controller.

For a specific platform, this functionality may be moved to `pci_config`(). However, because the Atlas and Malta boards support several "Core boards" with different system controller devices, it makes sense for these boards to keep the Core board specifics in a separate file.

Any fixed requirements for memory and I/O mapping set by the system controller should be setup by this function (setup '`bar_req`' array and adjust '`bar_count`'). The system controller data 'controller' must also be setup.

12.3.2 arch_pci_config_access()

This function is used for generating PCI configuration cycles (read or write). If no device responds, the function must return the error value:

```
ERROR_PCI_ABORT
```

12.3.3 arch_pci_lattim()

This function determines the latency timer value to be written to the PCI device.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

12.3.4 arch_pci_multi ()

This function extracts the "multi" field from the PCI configuration word 0xc (Word holding Bist/Header Type/ Latency Timer/Cache Line Size). The function treats the GT64120 system controller as a special case, because it is actually a multi-function device, and we only use function 0.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Chapter 13

CPU Reconfiguration

This chapter describes YAMON's support for reconfiguration of the CPU.

13.1 Overview

Some implementations of the MIPS families of CPUs allow configuration of the following parameters:

- Cache line size may be set to 0, which in effect causes corresponding cache (I- or D-cache) to be disabled.
- Cache associativity may be reduced.
- Number of cache lines per way may be reduced.
- If the CPU includes a TLB, the MMU may be configured as either TLB or "Fixed mapping".

This feature is present for the specific purpose of supporting configuration testing of the core in a lead vehicle, and is not supported in any other environment. Attempting to use this feature outside of the scope of a lead vehicle is a violation of the MIPS Architecture, and may cause unpredictable operation of the processor.

Configuration is performed by writing the following fields:

- *Config1*[24:22]: I-cache sets per way.
- *Config1*[21:19]: I-cache line size.
- *Config1*[18:16]: I-cache associativity.
- *Config1*[15:13]: D-cache sets per way.
- *Config1*[12:10]: D-cache line size.
- *Config1*[9:7]: D-cache associativity.
- *Config*[8]: TLB(0) / Fixed (1).

The above fields of the *CONFIG1* and *CONFIG* registers are by default Read Only. However, 4K/5K processor implementations may use *Config*[17] bit as "Write Control" (*WC*) bit. Setting *WC* makes the fields Read/Write.

YAMON probes whether the CPU is configurable (attempts to set *WC* bit and attempts to write fields). If one or more parameters are configurable, the shell command "`scpu`" is installed. This command may be used to configure the CPU. The new setting may be stored in the environment variable "`cpuconfig`", which on the Atlas board is also used for configuring the CPU following a reset. "`scpu`" and "`cpuconfig`" are further described in [1].

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Reconfiguring cache settings requires all ways of the corresponding cache (not just the reduced number of ways) to be reinitialized.

As an example, the function `sys_cpu_dcache_config`() (in file `arch/sys/cpu/sys_arch_cpu_s.S`) reconfigures the D-cache settings as follows:

- Disable interrupts by clearing *IE* bit of CP0 *STATUS* register.
- Flush D-cache.
- Start executing KSEG1 (uncached instructions).
- Set *Config*[17] (*WC*) bit.
- Restore max settings (i.e., hardware reset settings) for D-cache.
- Initialise D-cache.
- Write new D-cache settings.
- Clear *WC* bit.
- Restore CP0 *STATUS* register.
- Return to caller in KSEG0.

13.2 System Environment (SYSENV and ENV modules)

This section describes the implementation of environment variables.

YAMON maintains system-created environment variables and allows the user to create new environment variables (as described in [1]).

There are several layers involved in the implementation of environment variables:

- The flash driver (`drivers/flash/flash_strata.c`) writes and deletes the flash.
- The SYSENV module (implemented in `sysenv/sysenv.c`) manages records consisting of a fixed-length header and fixed-length data. It depends on the flash driver for writing/deleting the flash.
- The ENV module (implemented in `env` and `arch/env` directories) includes the functions used to create/modify/delete environment variables. They depend on SYSENV for managing the records.

The flash driver is accessed using the standard mechanism described in Section 3.3 “Driver API (IO Module)”.

SYSENV manages fixed records of 128 bytes, accessible through a logical index. The first 32-bit word of each record defines a control header, leaving 124 bytes for data (managed by ENV). Each control header contains four 8-bit fields:

1. status (free/in use)
2. logical index (assigned by ENV) used to reference record

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

3. size (byte count of user data).

4. checksum (simple 8-bit checksum of the data field bytes)

Each ‘SYSENV_write’ (defined in the include/sysenv_api.h) causes a record to be written into the environment FLASH at the next free location, maintained by SYSENV.

When the environment FLASH is full, i.e., no more free physical records are available, SYSENV performs a “garbage collection” procedure, which includes an erase of the entire environment FLASH to free up all “old” physical records no longer used.

SYSENV is accessed indirectly using SYSCON. The SYSCON object used for this is:

```
SYSCON_DISK_ENVIRONMENT_ID
```

The ENV module manages environment variables defined by two 0-terminated strings—one for the name, and one for the value. These strings are placed in a buffer, which is written to SYSENV for storage in flash.

ENV is initialised after a reset as described in Chapter 3, “Modules” on page 19 by calling the function env_init(). env_init() reads the available SYSENV records and creates/modifies system environment variables. Usually, system variables are not modified between resets, but if, for example, the RAM module is replaced with a module of different size, the environment variable “memsize” will be modified.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Copyright © 1999-2008 MIPS Technologies Inc. All rights reserved.

Appendix A

References

1. YAMON™ User's Manual
   MIPS document:MD00008

2. YAMON™ Errata
   MIPS document: MD00032

3. Atlas™ User's Manual
   MIPS document: MD00005

4. MIPS® Malta™ User's Manual
   MIPS document: MD00048

5. SEAD™ Basic RTL User's Manual
   MIPS document: MD00017

6. SEAD-2™ Basic Package Getting Started
   MIPS document: MD00062

Appendix B

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

Revision	Date	Description
01.00	00/01/07	Initial revision
01.01	00/02/08	Updated for YAMON 01.01
01.02	00/03/22	Updated copyright notice
02.00	00/09/11	Updated for YAMON 02.00
02.01	01/01/31	Document layout modified
02.02	01/07/27	Updated for YAMON 02.02
02.03	02/08/15	Updated for YAMON 02.03
02.04	02/11/21	Updated for YAMON 02.04
02.05	03/12/13	Updated for YAMON 02.05
02.06	04/03/24	Updated for YAMON 02.06
02.07	04/10/25	Updated for YAMON 02.07
02.08	05/03/21	Updated for YAMON 02.08
02.09	05/06/15	Updated for YAMON 02.09
02.10	05/10/14	Updated for YAMON 02.10
02.11	06/02/16	Updated for YAMON 02.11
02.12	06/0704	Updated for YAMON 02.12
02.14	07/11/12	Updated for YAMON 02.14
02.15	07/12/30	Updated for YAMON 02.15. No public release.
02.16	08/06/23	Updated for YAMON 02.16