Processor Initialization and Run-Time Services OSBOOT

Processor Initialization and Run-Time Services: OSBOOT, Release 3.3

Advanced Micro Devices, Inc.
5204 E. Ben White Blvd.
Austin, TX 78741–7399

Contents

About OSBOOT

Chapter 1

Overview of OSBOOT

Chapter 2

Using OSBOOT

Chapter 3

OSBOOT Directory and File Organization

Chapter 4

OSBOOT Bootstrap Module

Chapter 5

OSBOOT Trap Handlers

Chapter 6

HIF Run-Time Services

Chapter 7

Building OSBOOT or OSBOOT/DBG_CORE

Appendix A

Examples

Appendix B

Using the HIF IOCTL Service for Non-Blocking Reads

Appendix C

Defining a Trap to Switch to Supervisor Mode

Index

Figures and Tables

Figures

Tables

About OSBOOT

The Advanced Micro Devices (AMD®) **osboot** software contains the bootstrap module for the 29K™ Family of processors. The arithmetic instruction emulation routines for certain 29K Family instructions are part of the bootstrap module. **osboot** also includes a small kernel, called the HIF kernel, that provides the run-time support for application programs written in high-level languages. In addition, **osboot** isolates the processor dependencies from the application programs, offering a considerable decrease in design time for the developer.

osboot is ported to work with the AMD MiniMON29K™ debugger core, **dbg_core**. This manual describes the interactions between **osboot** and **dbg_core**.

osboot is used as the bootstrap program by the architectural simulator, **sim29**, and the instruction set simulator, **isstip**, in the AMD High C® 29K™ and MiniMON29K Software Development Kit.

This chapter provides an overview of the contents of the **osboot** documentation and describes the formatting conventions used within it.

OSBOOT Documentation

This documentation is written for advanced programmers using **osboot** to develop applications for the 29K Family of microprocessors and microcontrollers. For more information on these microprocessors and microcontrollers, see the list of suggested reference materials that follows.

About This Manual

Chapter 1: “Overview of OSBOOT” describes the major components of the **osboot** software and how it can be used to debug application programs using 29K Family microprocessors.

Chapter 2: “Using OSBOOT” describes the different ways **osboot** can be used. Models are used to show the various roles that **osboot** can play.

Chapter 3: “OSBOOT Directory and File Organization” describes the files used to make **osboot** and the directories in which they reside.

Chapter 4: “OSBOOT Bootstrap Module” describes the steps performed by the initialization module.

Chapter 5: “OSBOOT Trap Handlers” describes the Protection Violation, Unaligned Access, and Floating-Point Arithmetic Trap Handlers.

Chapter 6: “HIF Run-Time Services” describes the functions provided by the **osboot** HIF kernel, the run-time register and memory stack arrangements, and the implementations of the HIF services. The implementation of the HIF services is described in the context of the three models of **osboot** usage.

Chapter 7: “Building OSBOOT or OSBOOT/DBG_CORE” describes the steps involved in building **osboot** for simulators and the MiniMON29K debugger core, **dbg_core**. Both MS-DOS and UNIX environments are discussed. The chapter also describes the configurable link-time parameters of **osboot**.

Appendix A: “Examples” provides several examples of common test models using the **osboot** software, both with and without **dbg_core**.

Appendix B: “Using the HIF IOCTL Service for Non-Blocking Reads” provides an example of how a non-blocking read can be used to create an interactive menu that allows subsequent processing to continue while waiting for user input.

Appendix C: “Defining a Trap to Switch to Supervisor Mode” describes how to place a 29K Family microprocessor or microcontroller in supervisor mode.

“Index” provides an index to the manual.

Suggested Reference Material

The following reference documents may be of use to the **osboot** user:

- *Programming the 29K™ RISC Family*
  by Daniel Mann, P T R Prentice-Hall, Inc. 1994.
- *Am29000® and Am29005™ User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16914.
- *Am29030™ and Am29035™ Microprocessors User's Manual and Data Sheet*
  Advanced Micro Devices, order number 15723.
- *Am29040™ Microprocessor Data Sheet*
  Advanced Micro Devices, order number 18459.
- *Am29040™ Microprocessor User's Manual*
  Advanced Micro Devices, order number 18458.
- *Am29050™ Microprocessor Data Sheet*
  Advanced Micro Devices, order number 15039.
- *Am29050™ Microprocessor User's Manual*
  Advanced Micro Devices, order number 14778.
- *Am29200™ and Am29205™ RISC Microcontrollers Data Sheet*
  Advanced Micro Devices, order number 16361.
- *Am29200™ and Am29205™ RISC Microcontrollers User's Manual*
  Advanced Micro Devices, order number 16362.
- *Am29240™, Am29245™, and Am29243™ RISC Microcontrollers Data Sheet*
  Advanced Micro Devices, order number 17787.
- *Am29240™, Am29245™, and Am29243™ RISC Microcontrollers User's Manual*
  Advanced Micro Devices, order number 17741.
- Harbison, Samuel P. and Guy L. Steele, Jr.: *C: A Reference Manual, Second Edition*, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1987.
- *Host Interface (HIF) Specification*
  Advanced Micro Devices, order number 11539.

Standards and Conventions

Standards

This product complies with the following standards:

- ANSI C: American National Standards Institute C

  Conforms to the ANSI-approved document “Programming Language C,” document X3.159, 1989.

- COFF: AMD Common Object File Format

  Conforms to the AMD-augmented version of AT&T COFF, as described in the AMD *Common Object File Format (COFF) Specification.*

- HIF: AMD Host Interface

  Conforms to the AMD *Host Interface (HIF) Specification*.

- IEEE 754, 1985

  Conforms to the IEEE-approved standard for binary floating-point arithmetic.

- UDI: AMD Universal Debugger Interface

  Conforms to the AMD *Universal Debugger Interface (UDI) Specification.*

Conventions

- UNIX pathnames use a forward slash (/) to separate directories, while MS-DOS pathnames use a backslash (\). For brevity, only the DOS backslash is used when specifying pathnames. In some cases, code examples are specified as either for UNIX or MS-DOS environments and the correct slash is used.
- The following abbreviations may be used in this manual:
  - LSB least significant bit
  - LSW least significant word
  - MSB most significant bit
  - MSW most significant word
  - NaN not a number
  - QNaN quiet not a number
- In this manual, a data word signifies a 32-bit entity; a data halfword signifies a 16-bit entity.
- This manual uses the notational conventions shown in Table 0-1 (unless otherwise noted). These same conventions are used in all the 29K Family support products manuals.

Table 0-1. Notational Conventions

Symbol	Usage
Boldface	Indicates that characters must be entered exactly as shown. The alphabetic case is significant only when indicated.
Italic	Indicates a descriptive term to be replaced with a user-specified term.
`Typewriter face`	Indicates computer text input or output in an example or listing.
[ ]	Encloses an optional argument. To include the information described within the brackets, type only the arguments, not the brackets themselves.
{ }	Encloses a required argument. To include the information described within the braces, type only the arguments, not the braces themselves.
..	Indicates an inclusive range.
...	Indicates that a term can be repeated.
|	Separates alternate choices in a list—only one of the choices can be entered.
:=	Indicates that the terms on either side of the sign are equivalent.

Chapter 1

Overview of OSBOOT

AMD's **osboot** software, developed for the 29K Family of RISC microprocessors and microcontrollers, is a bootstrap program that resides in the ROM or instruction space of a 29K Family microprocessor target system. **osboot**'s main function is to control the execution states of the 29K Family microprocessor or microcontroller based on different hardware configurations. When a target system is reset, the microprocessor begins to fetch instructions from the **osboot** module in order to perform the target's initialization process.

There are two major components in the **osboot** module: a configurable bootstrap module and a small kernel for application programs. Each is described in this chapter. In addition, this chapter describes how **osboot** works with **dbg_core** and **montip**. Figure 1-1 illustrates the role of **osboot** in the High C 29K and MiniMON29K Software Development Kit.

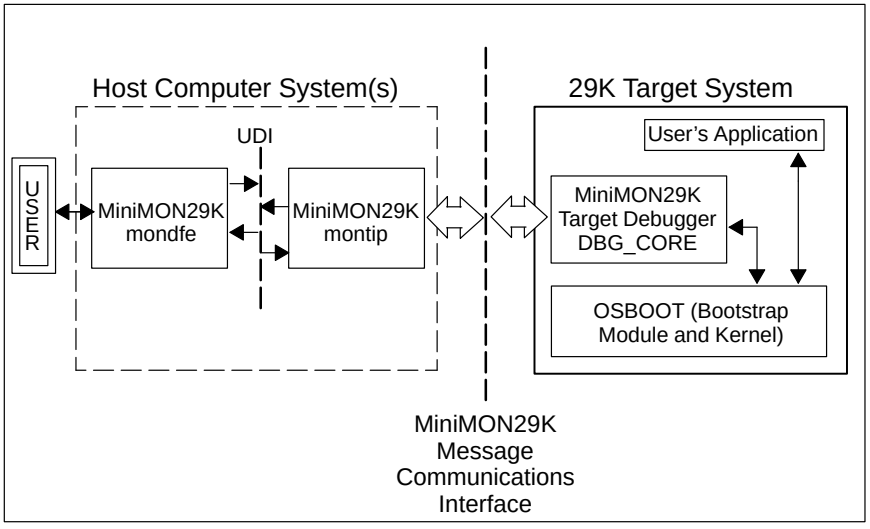

Figure 1-1. MiniMON29K Software Components

The Bootstrap Module

The bootstrap module of **osboot** provides the processor start-up functions, the optional floating-point emulation routines, the routines used to configure external memory dynamically, and the routines that provide for the RESET of any external or internal hardware that needs to be reset. These functions take control of the processor upon reset and perform the necessary tasks to initialize the target system to a defined state. The process of initializing the system at processor reset is called the *bootstrap* process or the *cold start* process. The bootstrap module must be loaded at offset 0x0 (zero) in the ROM address space or at offset 0x0 (zero) in the instruction address space from which the processor initiates its first instruction fetch.

Certain arithmetic instructions (mostly floating-point operations) of the 29K Family instruction set are not supported directly in the hardware of some members of the 29K Family. These instructions cause a trap when they are executed, and are emulated by the software trap routines. For more information on the trapped operations, see the “OSBOOT Trap Handlers” chapter.

The Kernel

The kernel takes control after the completion of the bootstrap process. It creates a single-threaded operating system environment in which programs can execute. It provides run-time support for application programs, especially for those written in high-level languages. In providing this support, the kernel implements the processor-specific services defined in AMD’s *Host Interface (HIF) Specification* (as opposed to file and I/O services). Processor-specific services are those services with a task/request number over 256. Hence, this kernel is called the HIF kernel of **osboot**.

MiniMON29K and OSBOOT

AMD's MiniMON29K software provides a means of debugging and testing 29K Family microprocessor application programs. The two major sections of the MiniMON29K software discussed in this manual are the debug core module (**dbg_core**) and the MiniMON29K target interface process (**montip**). **osboot** provides a simple operating system that can be linked with **dbg_core** to provide an environment to develop, execute, and debug embedded applications based on 29K Family microcontroller or microprocessor targets. The following sections provide an overview of this environment.

About DBG_CORE

dbg_core is the *target-resident* debugger module of the MiniMON29K product. "Target-resident" means that **dbg_core** resides on the target hardware. Linking **osboot** with **dbg_core** creates an environment in which the programmer can develop, execute, and debug 29K Family microprocessor embedded applications on the target itself.

About MONTIP

montip is the *host-resident* debugger module of the MiniMON29K product. "Host-resident" means that **montip** resides on a PC or UNIX workstation (rather than the target hardware). **montip** and **dbg_core** share a common message system and can understand each other's communications. The communication system used by **montip** is media-independent, but has been implemented to support serial ports and parallel ports.

montip is implemented according to AMD's Universal Debugger Interface (UDI) specification. To use **montip** for debugging, it must be connected to a UDI-compliant Debugger Front End (DFE). DFEs provide the user interface for the debugging process. Because of the UDI standard, the DFE is isolated from the execution vehicle. In this way, the same debug tools can be used with a software target, emulator, or monitor. UDI-compliant DFEs currently available include **mondfe** (provided with the MiniMON29K product), XRAY29K™, UDB™, and gdb. Other DFEs are currently in development.

The OSBOOT/DBG_CORE/MONTIP/DFE Environment

Within the MiniMON29K product, **osboot** is linked with **dbg_core** to provide an environment to develop, execute, and debug embedded applications on targets based on 29K Family microprocessors. The following example (illustrated in Figure 1-1 on page 1-1) provides a high-level overview of the interactions between **osboot**, **dbg_core**, **montip**, and the DFE during the debugging process. In Figure 1-1, **mondfe** is used as the DFE. A more thorough example is provided in the "Using OSBOOT" chapter.

Suppose a user wants to view the contents of a register on the target's 29K Family microprocessor. Using a UDI-compliant DFE on the host computer system, the user requests the contents of a register on the 29K Family target. This request is passed via UDI to **montip** (still on the host computer system). Upon receiving the request from the DFE, **montip** passes the request via MiniMON29K messages on to the **dbg_core** module on the target. In turn, the **dbg_core** module communicates with **osboot** through the debug channel to retrieve the results of the request. The results are passed back to the user's DFE by reversing the process. Executing a program or setting a breakpoint from the DFE is done in a similar manner.

The advantage of separating UDI and MiniMON29K messages is that UDI messages can travel over (for example) an Ethernet network (local- or wide-area), while MiniMON29K messages travel over a serial link (or parallel link or shared memory) to the target board. This flexibility allows communications to take place over the most convenient available media.

Chapter 2

Using OSBOOT

The **osboot** software is used as the bootstrap program by the architectural simulator, **sim29**, and the instruction set simulator, **isstip**, of AMD's High C 29K and MiniMON29K Software Development Kit. **osboot** is also a simple operating system which is ported to the MiniMON29K debugger core, **dbg_core**. The design of **osboot** is such that all or portions of it can be linked with application programs to produce stand-alone systems.

The following sections briefly discuss several models using the **osboot** software:

OSBOOT/Simulator Model

The simplest implementation among those described in this chapter is the OSBOOT/Simulator Model, shown in Figure 2-1. Because a simulator is a software target (that is, no actual hardware is involved), **osboot** uses a stripped-down version of the bootstrap module. In the model shown in Figure 2-1, **osboot** neither implements routines to configure external memory dynamically, nor does it initialize the external communications interface. In this model, when the application program requests an I/O service, the simulator intercepts the I/O requests to the HIF kernel and performs the necessary operations using the services of its host operating system.

AMD provides two different simulators: the 29K Family architectural simulator (**sim29**, which provides timing and statistical information about the application), and the target interface process instruction set simulator (**isstip**, which runs the program without timing or statistical information). “Building OSBOOT for Simulators,” on page 7-4, discusses how to build the **osboot** module for simulators.

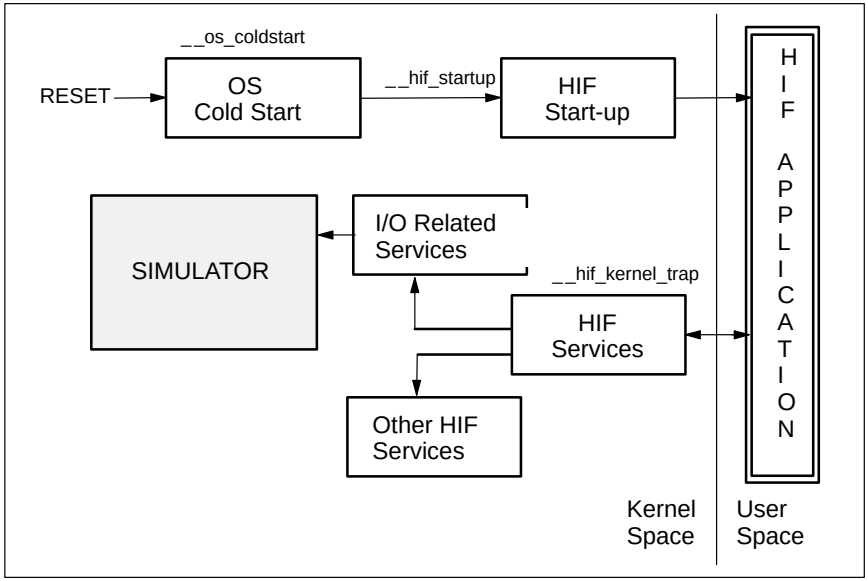

Figure 2-1. OSBOOT/Simulator Model

OSBOOT/DBG_CORE Model

In the OSBOOT/DBG_CORE model shown in Figure 2-2, **osboot** is linked with the MiniMON29K debugger core, **dbg_core**, to provide an environment to debug, develop, and execute embedded applications on targets based on the 29K Family microprocessors.

Figure 2-2 shows the **osboot** bootstrap module as modified to initialize the **dbg_core** message system and to install the trap handlers provided by **dbg_core**. After the completion of the bootstrap process, the bootstrap module transfers control temporarily to **dbg_core** to initialize the debug core by calling the **dbg_control()** function inside **dbg_core**.

The HIF kernel of **osboot** is invoked when the call to the **dbg_control()** function returns control to **osboot**. The **dbg_control()** function returns the application program information in general purpose registers **gr96** through **gr103**. The HIF kernel uses the return values and prepares an operating environment for the application program.

As shown in Figure 2-2, the HIF kernel performs the I/O operations requested by the application program using the **dbg_core** message system. It sends a UDI-compliant MiniMON29K message to the MiniMON29K target interface process, **montip**, over a communication interface (such as a serial interface or shared memory). The message is received by **montip** and is processed on an intelligent host. **montip** sends a MiniMON29K message back to the HIF kernel that contains the results of the I/O operation. Refer to the “HIF Run-Time Services” chapter for a more detailed explanation.

In this model (shown in Figure 2-2), the **osboot** and **dbg_core** modules reside in the ROM units of the target. The program to be tested and debugged is downloaded to the RAM units of the target. To test and debug an application program in this manner requires that the target hardware meet the following minimum requirements:

- There must be a communication channel between the **montip** host and the target. A PC-hosted **montip** supports both serial and parallel interfaces. A UNIX workstation-hosted **montip** supports only serial interfaces.
- The ROM unit of the target must have at least 64 Kbyte of memory.
- The RAM units of the target must have at least 16 Kbyte of memory.

“Building OSBOOT/DBG_CORE for Hardware Platforms” on page 7-8 describes how to build the OSBOOT/DBG_CORE model.

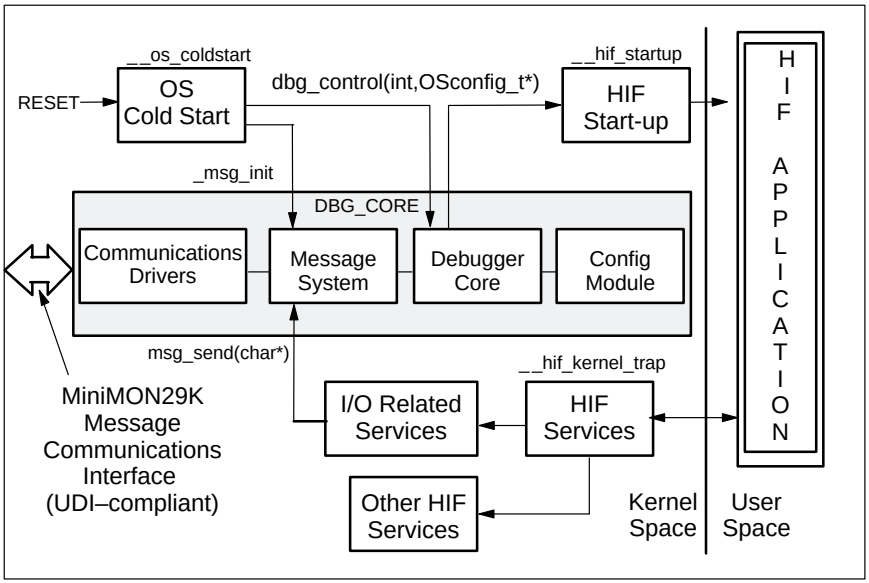

Figure 2-2. OSBOOT/DBG_CORE Model

Stand-Alone OSBOOT/Application Model

In the Stand-Alone OSBOOT/Application Model shown in Figure 2-3, the application program is linked with **osboot** and executed from the ROM memory space on the target system. Program information such as entry point, stack requirements, and execution mode must be provided at compile time. The HIF kernel requires this information to establish an operating environment for the application. In this model, the HIF kernel uses its own communications drivers or those provided by the application program to perform the I/O operations. The shaded boxes shown in Figure 2-3 represent the modules specific to the stand-alone **osboot** software.

This model is usually used in the final stages of testing, when the target hardware and the application software are fully debugged and ready to be released.

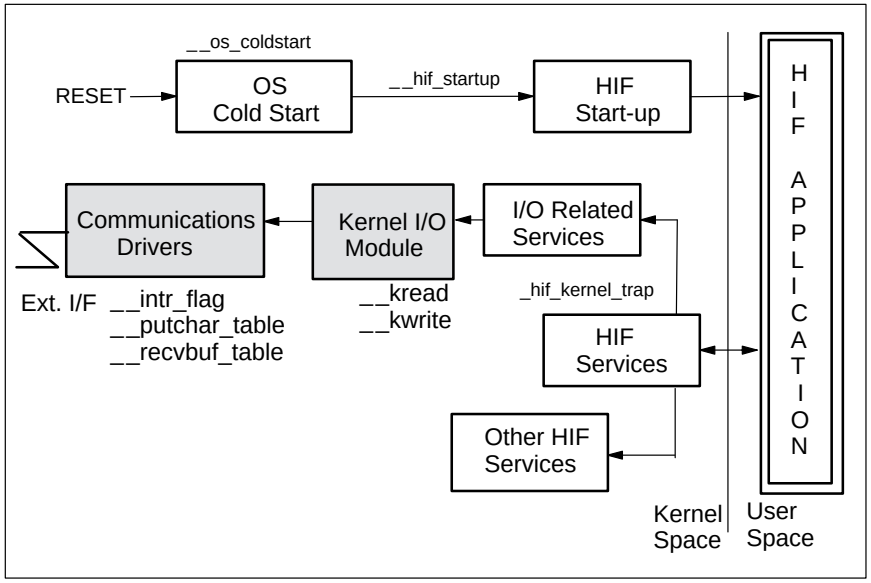

Figure 2-3. Stand-Alone OSBOOT/Application Model

Stand-Alone OSBOOT/DBG_CORE/Application Model

The Stand-Alone OSBOOT/DBG_CORE/Application Model (shown in Figure 2-4) provides the same functionality included in the Stand-Alone OSBOOT/Application Model (described previously), with the addition of the debugging and testing capabilities provided by the MiniMON29K software's **dbg_core** module.

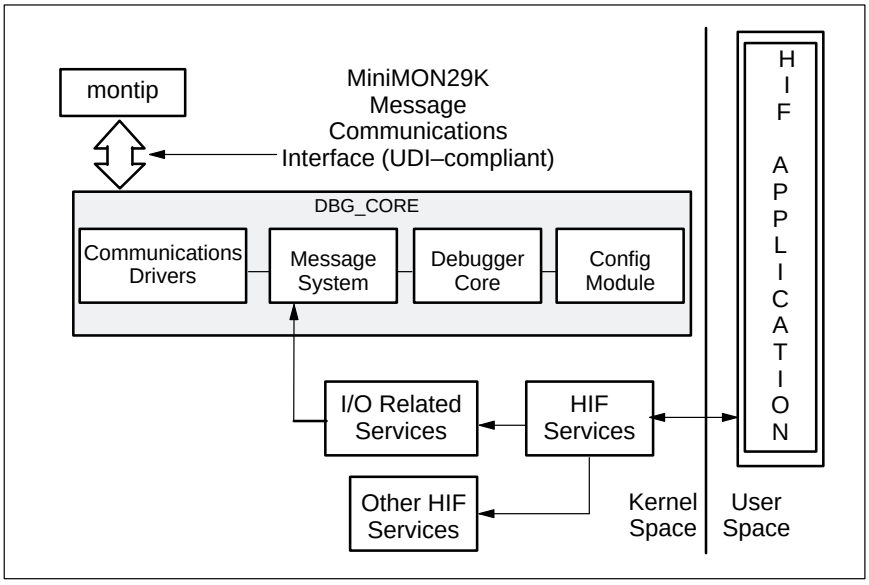

Figure 2-4. Stand-Alone OSBOOT/DBG_CORE/Application Model

In this model, the application program is linked with the **osboot/dbg_core** module and executed from ROM memory space on the target hardware. As shown in Figure 2-4, the HIF kernel performs the I/O operations requested by the application program using the **dbg_core** message system. It sends a UDI-compliant MiniMON29K message to the MiniMON29K target interface process, **montip,** over a communication interface (such as a serial interface, or shared memory). The message is received by **montip** and is processed on an intelligent host. **montip** sends a MiniMON29K message that contains the results of the I/O operation back to the HIF kernel. Refer to the "HIF Run-Time Services" chapter for a more detailed explanation.

Chapter 3

OSBOOT Directory and File Organization

The complete **osboot** source code is provided for educational and customization purposes. However, porting **osboot** to new targets often only requires that changes be made to the linker command file – no changes need to be made to source files.

The sources of **osboot** are located in the **29k\src\osboot** directory of the AMD tree structure. This is the top-level directory of the **osboot** sources.-Figure 3-1 on page 3-3 shows the files in the **osboot** directory.

The **29k\src\osboot** directory contains the following make files and MS-DOS batch files:

- UNIX make files:
  - **makefile.os** to build **osboot** for simulators or to build a relocatable **osboot** module to link with a stand-alone debugged application
  - **makefile.mon** to build **osboot/dbg_core** for debugging applications
- MS-DOS batch files:
  - **makeosb.bat** to build **osboot** for simulators or to build a relocatable **osboot** module to link with a debugged stand-alone application
  - **makemon.bat** to build **osboot/dbg_core** for debugging applications

The linker command files used by the make files and MS-DOS batch files to produce relocatable objects and absolute images of **osboot** and **osboot/dbg_core** for different targets are shown in Table 3-1.

The linker command filename extensions have the following meanings:

- **.inc** files produce relocatable **osboot**
- **.mon** files produce a relocatable image of **osboot** for linking with **dbg_core**
- **.lnk** files produce absolute objects of **osboot** or **osboot/dbg_core**

The **29k\src\osboot** directory contains two subdirectories, **boot** and **traps,** which are explained in the sections starting on pages 3-4 and 3-6, respectively.

Table 3-1. OSBOOT Linker Command Files

Target Name	Linker Command File		
	Relocatable osboot	Absolute	Relocatable osboot–dbg_core
AMD's EB29030	eb29030.inc	eb29030.lnk	eb29030.mon
AMD's EB29K	eb29k.inc	eb29k.lnk	eb29k.mon
AMD's EZ030	ez030.inc	ez030.lnk	ez030.mon
Laser29K-030 board	la29030.inc	la29030.lnk	la29030.mon
Laser29K-200 board	la29200.inc	la29200.lnk	la29200.mon
YARC's ATM sprinter	lcb29k.inc	lcb29k.lnk	lcb29k.mon
Netrom	netrom.inc	netrom.lnk	netrom.mon
AMD's PCEB29K	pceb.inc	pceb.lnk	pceb.mon
AMD's SA29030	sa29030.inc	sa29030.lnk	sa29030.mon
AMD's SA29200	sa29200.inc	sa29200.lnk	sa29200.mon
AMD's SA29205	sa29200.inc	sa29200.lnk	sa29200.mon
AMD's SE29240	sa29240.inc	sa29240.lnk	sa29240.mon
AMD's SE29040	se29040.inc	se29040.lnk	se29040.mon
STEP's STEB29K	steb.inc	steb.lnk	steb.mon
YARC's Rev 8	yarcrev8.inc	yarcrev8.lnk	yarcrev8.mon
Instruction Set Simulator (**isstip**) or Architectural Simulator (**sim29**)	sim.inc sim245.inc	sim00x.lnk sim03x.lnk sim050.lnk sim20x.lnk sim24x.lnk	

NOTE: Some of these boards are no longer available commercially, but are still in use. Note also that the names of some boards' linker command files do not correspond directly to the board's name.

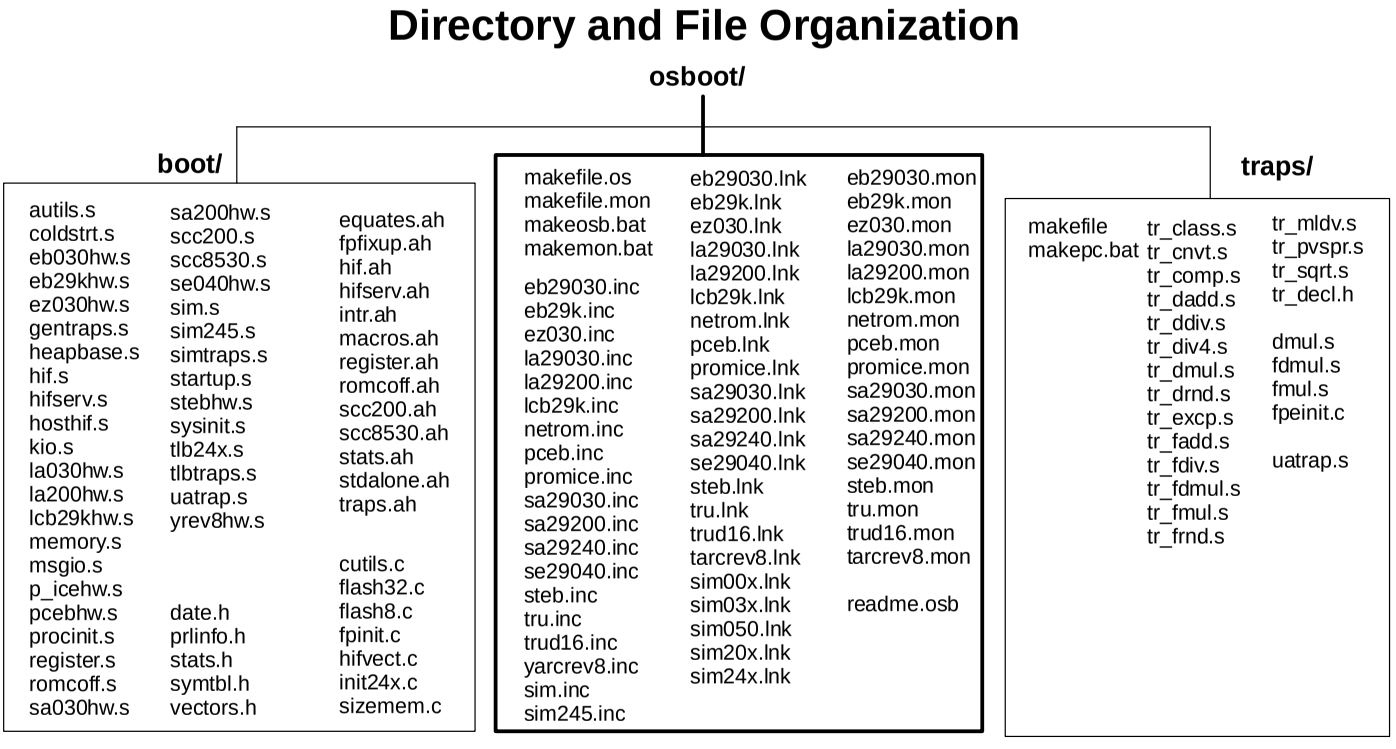

Figure 3-1. OSBOOT Directory and File Organization

The boot Subdirectory

The **boot** subdirectory contains the source files that implement the bootstrap module and the HIF kernel of **osboot**.

The filename extensions of the source files in the **29k/src/osboot/boot** directory are explained below:

- **.s** indicates assembly language functions
- **.ah** indicates assembler header files
- **.c** indicates C functions
- **.h** indicates C header files

Table 3-2 lists the source files in the **boot** subdirectory and the function(s) that they implement.

Table 3-2. OSBOOT Source Files under boot Subdirectory

Source Filename	Function Name(s)
startup.s	__os_startup
sim.s, sim245.s	__os_coldstart, __os_raminit, __os_memset
coldstrt.s	__os_coldstart, __os_raminit
procinit.s	__os_procinit, __os_am29000_init, __os_am29005_init __os_am29050_init, __os_am29030_init, __os_am29035_init, __os_am29200_init, __os_am29205_init, __os_am29240_init
init24x.c	__os_am2924x_init
memory.s	__os_sizememory, __os_memset
sizemem.c	__os_sizemem29k
sysinit.s	__os_sysinit
gentraps.s	__os_warn_mm_trap, __os_unexpected_trap, __os_illegal_op_trap, __os_trace_trap
simtraps.s	__os_warn_mm_trap, __os_raminit, __os_memset, __os_illegal_op_trap, __os_unexpected_trap, __os_trace_trap __hif_hosthif
uatrap.s	__os_ua_trap

Source Filename	Function Name(s)
tlbtraps.s	__os_uiTLBmiss_trap, __os_udTLBmiss_trap
tlb24x.s	__os_uiTLBmiss24x_trap, __os_udTLBmiss24x_trap
register.s	defines the register mnemonics used
hif.s	__hif_startup, __hif_flushTLB, __hif_reboot, __hif_spillhandler, __hif_fillhandler
hifserv.s	__hif_timer_trap, __hif_spill_trap, __hif_fill_trap, __hif_kernel_trap
hosthif.s	__hif_hosthif
msgio.s	__hif_hosthif (uses **dbg_core** message system)
eb030hw.s	__os_initcomm for EB29030 card
ez030hw.s	__os_initcomm for EZ030 card
eb29khw.s	__os_initcomm for EB29K card
la030hw.s	__os_initcomm for Laser29K–030 board
la200hw.s	__os_initcomm for Laser29K–200 board
lcb29khw.s	__os_initcomm for YARC ATM Sprinter card
pcebhw.s	__os_initcomm for PCEB29K card
sa030hw.s	__os_initcomm for SA29030 board
sa200hw.s	__os_initcomm for SA29200, SE29240, and SA29205 boards
se040hw.s	_ os_initcomm for SE29040 board
stebhw.s	__os_initcomm for STEB board
yrev8hw.s	__os_initcomm for YARC Rev 8 card
kio.s	__kread, __kwrite
scc200.s	__scc200_init, __scc200_putchar, __scc200_getchar
scc8530.s	__scc8530_init, __scc8530_putchar, __scc8530_getchar, __scc8530_intr

The traps Subdirectory

The **traps** subdirectory contains the source files that implement the trap handlers for arithmetic operations which the processor hardware does not support directly. It also contains the sources for the Protection Violation trap handler, which are used to access the virtual special-purpose registers.

The following files are also included in the **traps** subdirectory to build a library of trap routines:

- UNIX make file: **makefile**
- MS-DOS batch file: **makepc.bat**

The bootstrap module of **osboot** is linked with the trap routine library. The necessary trap handlers are installed during the bootstrap process.

The trap handler source files and floating-point trap routine(s) are listed in Table 3-3. The filename extensions are explained below:

- **.s** indicates assembly language functions
- **.h** indicates assembler header files

Table 3-3. Arithmetic Trap Handler Source Files

Files	Routines
fpeinit.c	Floating-point emulation handlers
tr_class.s	CLASS instruction emulation routine
tr_cnvt.s	CONVERT instruction emulation routine
tr_comp.s	FEQ, FGE, FGT, DEQ, DGE and DGT instruction emulation routines
tr_dadd.s	DADD and DSUB instruction emulation routines
tr_ddiv.s	DDIV instruction emulation routines
tr_div4.s	DIVIDE and DIVIDU instruction emulation routines using Am29240™ processor INTE register
tr_dmul.s	DMUL instruction emulation routines using 32x32 bit multiplier
tr_drnd.s	Double-precision round/range check routines
tr_excp.s	Floating-point exception trap routine
tr_fadd.s	FADD and FSUB instruction emulation
tr_fdiv.s	FDIV instruction emulation routine
tr_fdmul.s	FDMUL instruction emulation routine
tr_fmul.s	FMUL instruction emulation routine
tr_frnd.s	Single-precision round/range check routine
tr_mldv.s	MULTIPLY, MULTIPLU, MULTM, MULTMU, DIVIDE and DIVIDU instruction emulation routines
tr_pvspr.s	Protection violation trap handler, **__os_29kpv_trap**
tr_sqrt.s	SQRT instruction emulation
uatrap.s	Unaligned access trap handler

NOTE: The register definitions of the mnemonics used in the files listed above are in **register.s** and **register.ah** files in the **boot** subdirectory.

Chapter 4

OSBOOT Bootstrap Module

When the target system is powered on (RESET), the processor begins to fetch and execute instructions from offset 0x0 in ROM space or instruction space. These instructions initialize the processor and the external target system to a defined state. This process of bringing up the system from RESET to a defined state is called the *bootstrap process* or the *cold start* process. The tasks performed during the bootstrap process are implemented in the **osboot** bootstrap module.

Figure 4-1 illustrates in a left-to-right sequence the tasks performed by the **osboot** bootstrap module. The implementation and the associated file(s) of each task are described in the following sections, as listed in Figure 4-1.

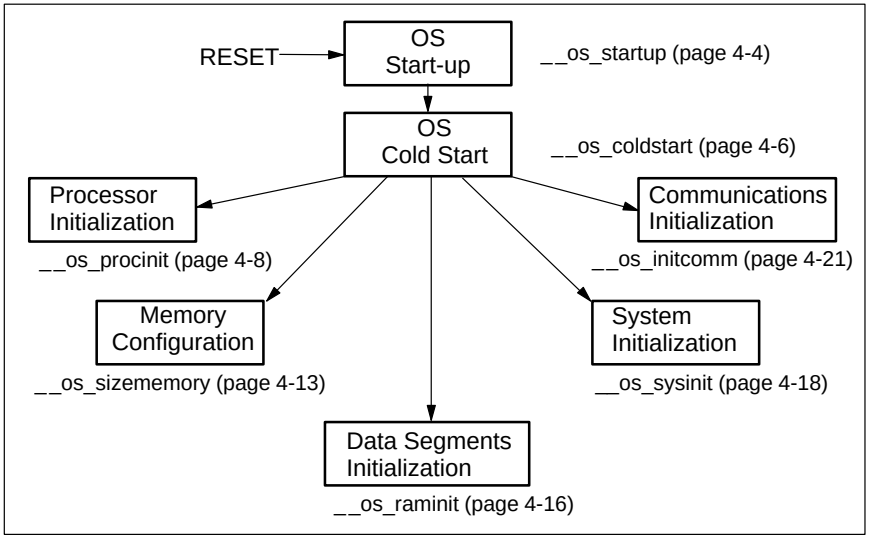

Figure 4-1. OSBOOT Bootstrap Module

OSBOOT Global Register Usage

Table 4-1 shows the registers used by **osboot** to implement the floating-point emulation routines and the HIF kernel services. The files **register.s** and **register.ah** define the symbolic register names and their associated physical processor registers. The **register.s** file, when assembled and linked with other **osboot** files, provides a global linkage to the registers listed in Table 4-1. The **register.ah** file contains external references to each of the registers defined in the **register.s** file, and is included in **osboot** source files containing assembler programs.

Table 4-1. Register Definitions

Register	Symbols Used	Description
gr64	it0, TempReg0	Interrupt or trap handler temporary register
gr65	it1, TempReg1	Interrupt or trap handler temporary register
gr66	it2, TempReg2	Interrupt or trap handler temporary register
gr67	it3, TempReg3	Interrupt or trap handler temporary register
gr68	kt0, TempReg4	Kernel temporary register
gr69	kt1, TempReg5	Kernel temporary register
gr70	kt2, TempReg6	Kernel temporary register
gr71	kt3, TempReg7	Kernel temporary register
gr72	kt4, TempReg8	Kernel temporary register
gr73	kt5, TempReg9	Kernel temporary register
gr74	kt6, TempReg10	Kernel temporary register
gr75	kt7, TempReg11	Kernel temporary register
gr76	kt8, TempReg12	Kernel temporary register
gr77	kt9, TempReg13	Kernel temporary register
gr78	kt10, TempReg14	Kernel temporary register
gr79	kt11, TempReg15	Kernel temporary register
gr89	TimerExt	Timer extension register
gr90	SpillAddrReg	Spill trap handler address

Register	Symbols Used	Description
gr91	FillAddrReg	Fill trap handler address
gr92	FPStat3	Floating-point trap handler static register
gr93	FPStat2	Floating-point trap handler static register
gr94	FPStat1	Floating-point trap handler static register
gr95	FPStat0	Floating-point trap handler static register

NOTE: Please refer to the comments in the **register.ah** file in the **29k/src/osboot/boot** directory for further information on register usage.

OS Start-Up and WARN Trap Handler

File

`startup.s`

The file **startup.s** implements the **osboot** start-up function that takes control of the processor on RESET. It also contains the processor WARN trap handler, which is also the Monitor Mode trap handler for the Am29050™ microprocessor. The functions implemented in the **startup.s** file are contained in a single text section called Reset. The code below shows the contents of the Reset text section. This section must be loaded and ordered at offset 0x0 in the ROM address space or in the instruction/data memory in the linker command file.

Functions

`__os_startup, address16`

External Functions

`__os_coldstart, __os_warn_mm_trap`

Reset Text Section

```
         .extern  __os_coldstart
         .extern  __os_warn_mm_trap
         .sect    Reset, text
         .use     Reset
         .global  __os_startup
__os_startup:
         jmp      __os_coldstart
         nop
         nop
         nop
address16:
         jmp      __os_warn_mm_trap
         nop
         nop
         nop
         .text
```

Description

The **__os_startup** function is the **osboot** start-up function. It is executed from offset 0x0 when the processor is powered on. It invokes the **__os_coldstart** routine which controls the rest of the cold start process. The **__os_startup** routine must contain no more than four instructions, because it is immediately followed by **address16**, which is the WARN trap handler or the Monitor Mode trap handler of the Am29050 processor.

address16 is the WARN trap handler or the Monitor Mode trap handler for the Am29050 microprocessor and must be aligned at offset 0x10 in ROM address space or in instruction/data memory address space. When a WARN trap or Monitor Mode trap occurs, the **__os_warn_mm_trap** routine is executed using a direct jump (jmp) instruction.

OS Cold Start

Files

`sim.s, coldstrt.s, sim245.s`

The files **sim.s, sim245.s** and **coldstrt.s** implement the **__os_coldstart** function, which controls the cold start process for software targets (such as simulators) and for hardware targets. The **__os_coldstart** function performs a number of tasks to initialize the target system to a defined state. These tasks are implemented as separate functions which are called during the cold start process.

Link-Time Constant

`DMemStart`

DMemStart is a link-time constant that must be defined in the linker command file (with **.lnk** file extension). It defines the start of the external data memory region of the specific target system. **DMemStart** is used as the starting address for the dynamic memory sizing performed to configure the external memory systems. It is used as the base address for the placement of the vector table. The special-purpose register of the processor,Vector Area Base (VAB), is initialized with this constant.

Function

`__os_coldstart`

External Functions

```
__os_procinit
__os_sizememory
__os_memset
__os_sysinit
__os_check027
__os_fpinit
__os_initcomm
```

Description

The function **__os_coldstart** controls the cold start process. It is executed by the OS Start-Up module after processor RESET, as shown in Figure 4-1 on page 4-1. It sets up a pseudo register stack to avoid accidental spilling and filling of registers before calling different functions. The functions supplied with the **osboot** software are designed such that spilling and filling of registers is not required.

The special purpose register of the processor, VAB, is initialized with the value of the **DMemStart** link time constant.

Several tasks are performed during the cold start process. The following list outlines these tasks, appearing in the order in which they are executed. Each of these tasks is described in detail in the following sections.

1. Processor Initialization—Initializes the processor special registers and peripheral registers to a defined state. Page 4-8.
2. Memory Configuration—Dynamically sizes the available external DRAM memory and programs the DRAM Controller accordingly. Page 4-13.
3. Data Segments Initialization—Initializes the data segments in memory, clears the BSS sections, and, if in a standalone environment, transcribes initialized data sections from ROM to data memory. Page 4-16.
4. System Initialization—Initializes the vector table of the system and saves target configuration information in memory. Page 4-18.
5. Communications Initialization—Initializes the communications interface used by the target system. This initializes the **dbg_core** message system in the OSBOOT/DBG_CORE Model. Page 4-21.

The completion of the tasks listed above marks the end of the cold start process. In the OSBOOT/DBG_CORE Model, control is temporarily transferred to the **dbg_core** by calling the **dbg_control()** function. When **dbg_control()** returns, the HIF kernel is started up.

Processor Initialization

Files

`procinit.s, init20x.c, init24x.c`

The **procinit.s** file implements the function that performs processor initialization at cold start. During processor initialization, the timer of the processor is disabled and some of the special registers of the processor are initialized to defined values. In the case of the 29K Family microcontrollers, the peripheral registers are also initialized and the devices are left disabled.

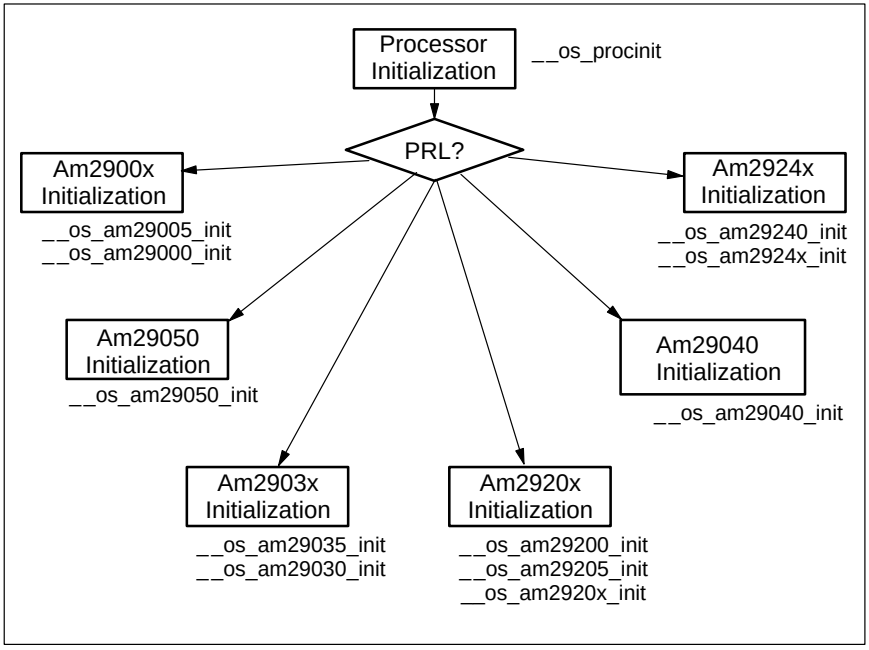

Figure 4-2. Processor Initialization

Function

`__os_procinit`

Auxiliary Functions

```
__os_am29000_init
__os_am29005_init
__os_am29030_init
__os_am29035_init
__os_am29040_init
__os_am29050_init
__os_am29200_init
__os_am29205_init
__os_am2920x_init
__os_am29240_init
__os_am2924x_init
```

Link-Time Constants

```
DRCT_VALUE
init_CPS
RMCF_VALUE
RMCT_VALUE
```

DRCT_VALUE contains the value to initialize the DRAM Controller peripheral register of the 29K Family microcontrollers for the target system configuration.

init_CPS contains the value to initialize the processor's Current Processor Status (CPS) register for the cold start process.

RMCF_VALUE contains the value to initialize the ROM Configuration peripheral register of the 29K Family microcontroller for the target system configuration.

RMCT_VALUE contains the value to initialize the ROM Controller peripheral register of the 29K Family microcontrollers for the target system configuration.

Description

The **__os_procinit** function is called from **__os_coldstart** during the cold start process. It initializes the contents of some processor special registers with defined values. The special registers initialized are as follows:

- Current Processor Status (CPS) register
- Timer Reload (TMR) register
- Register Bank Protect (RBP) register
- Configuration (CFG) register
- Memory Management Unit (MMU) register

The CPS register is initialized with the **init_CPS** value defined at link time. The TMR register is cleared and the timer is disabled. The register bank protect register is set to zero to protect no registers.

After initializing the CPS, TMR, and RBP registers, **__os_procinit** examines the PRL field of the CFG register to determine the processor type. The PRL field of the CFG register is the eight most-significant bits of the register. Based on the PRL value found, the appropriate routine to perform processor-specific initialization is called. Table 4-2 lists processor PRL fields and their function names.

Table 4-2. Processor PRL Fields

Processor	PRL	Values	Function Name(s)
	31–27	26–24	
Am29000	0	x	__os_am29000_init
Am29005	0	x	__os_am29005_init
Am29050	2	x	__os_am29050_init
Am29035	4	x	__os_am29035_init
Am29030	4	x	__os_am29030_init
Am29200	5	x	__os_am29200_init
Am29205	58	x	__os_am29205_init
Am29240	6	x	__os_am29240_init
Am29040	7	x	__os_am29040_init

NOTE: **x** indicates these bit positions do not matter

__os_am29000_init and __os_am29005_init

The **__os_am29000_init** and **__os_am29005_init** functions perform the following tasks:

- Initialize the MMU register for 8K page size (Am29000 only).
- Set the process ID field to 0x1.
- Disable the BTC (branch target cache) for processor revisions A and B (Am29000 only).
- Set the VF and DW bits of the CFG register.
- Determine if the external memory system supports byte accesses.
- Update the DW bit of the CFG register.

__os_am29050_init

The **__os_am29050_init** function performs the following tasks:

- Initializes the MMU register for 8K page size.
- Sets the process ID field to 0x1.
- Sets the VF and DW bits of the CFG register.
- Determines the byte-writability of the external memory system.
- Updates the DW bit of the CFG register.

__os_am29030_init and __os_am29035_init

The **__os_am29030_init** and **__os_am29035_init** functions perform the following tasks:

- Set the CFG register to disable the cache.
- Initialize the MMU register for 8K page size.
- Set the process ID field to 0x1.

NOTE: The **__os_am29030_init** and **__os_am29035_init** functions are defined in a separate text section named **Sect030** so that they can be aligned on a quad-word boundary at the time of linking.

__os_am29040_init

The **__os_am29040_init** function performs the following tasks:

- Sets the CFG register to disable both caches.
- Initializes the MMU register for 4K page size for both sets of TLBs.
- Sets the process ID field to 0x1.

__os_am29200_init, and __os_am29205_init

The **__os_am29200_init**, and **__os_am29205_init** functions perform the following tasks:

- Initialize the ROM Controller register with the value of the **RMCT_VALUE** link-time constant.
- Initialize the ROM Configuration register with the value of the **RMCF_VALUE** link-time constant.

- Initialize the DRAM Controller register with the **DRCT_VALUE** link-time constant defined in the linker command file for the target system configuration.
- Call **__os_am2920x_init** to perform the rest of the initialization.

__os_am29240_init

The **__os_am29240_init** function performs the following tasks:

- Sets the CFG register to disable both caches.
- Initializes the MMU register for 1K page size for both sets of TLBs.
- Sets the process ID field to 0x1.
- Initializes the ROM Controller register with the value of the **RMCT_VALUE** link-time constant.
- Initializes the DRAM Controller register with the **DRCT_VALUE** link-time constant defined in the linker command file for the target system configuration.
- Calls **__os_am2924x_init** to perform the rest of the initialization.

Memory Configuration

Files

`memory.s, sizemem.c, size200.c`

The **memory.s** file implements the function that dynamically sizes the external DRAM memory region using a non-destructive memory sizing technique. It programs the DRAM Configuration register on the 29K Family microcontrollers according to the memory available and returns the total size of memory available to the caller—in this case, the cold start process.

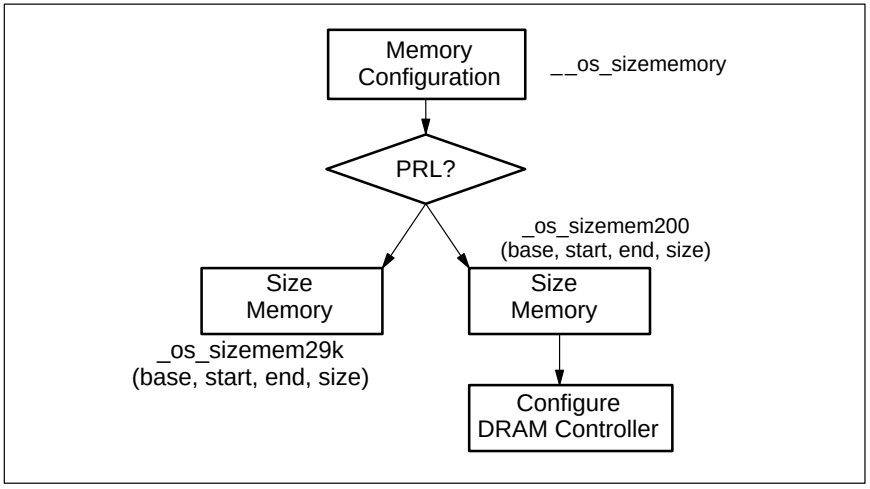

Figure 4-3. Memory Configuration

Function

`__os_sizememory`

Auxiliary Functions

```
_os_sizemem29k(long *base, long *start, long *end, int size)
_os_sizemem200(long *base, long *start, long *end, int size)
```

Link-Time Constants

```
DMemStart
DMemSize
```

DMemStart defines the base address of the external DRAM memory.
DMemSize is the maximum size of memory on the target system.

Description

The **__os_sizememory** function, defined in the **memory.s** file, is called during the cold-start process. Based upon the value of the PRL field in the configuration register (CFG), the function calls either **__os_sizemem29k** for the Am29000 Family of microprocessors, or **__os_sizemem200** for the Am29000 Family of microcontrollers and passes them the necessary input arguments.

The arguments passed to the **__os_sizemem29k** and **__os_sizemem200** functions are derived from the **DMemStart** and **DMemSize** link-time constants defined in the linker command file for the target system configuration.

The **__os_sizemem29k** function is defined in the **sizemem.c** file and is called from **__os_sizememory** and from **__os_sizemem200** to size a segment of external memory. The first parameter, **base**, gives the base address of the data memory being sized. The second parameter, **start,** gives the memory address from which to begin the sizing algorithm. The third parameter, **end**, gives the memory address at which to terminate the sizing algorithm. The address is computed by adding the **DMemSize** value to the **DMemStart** value. The fourth parameter, **size**, gives the size of the unit memory chip. This value is used as an increment in the sizing algorithm.

Sizing Algorithm

This procedure is a loop that starts with the test address initialized to the address contained in the **start** parameter. It stores the test address at the test location, writes the **base** address in the **base** location, and reads back the value from the test location. It then compares the value read with the test address. If the values are the same, then the next test location is computed by adding the memory chip size specified in the **size** parameter, and the loop is executed again. The total size of memory found is incremented by the memory chip size. If the values are not the same, the loop exits, and the total memory found thus far is returned.

The **__os_sizemem200** function is defined in the **sizemem.c** file and is called by **__os_sizememory** to size the external memory and program the DRAM configuration register of the 29K Family microcontrollers. The **__os_sizemem200** function takes the same input parameters as that of **__os_sizemem29k**. It sizes each DRAM bank by calling **__os_sizemem29k**, and computes the ASEL fields and AMASK fields for the bank sizes found. It arranges the banks in descending order by size and programs the DRAM configuration register.

Data Segments Initialization

Files

`coldstrt.s, sim.s, sim245.s`

Global Symbol

`RAMInit`

The **sim.s**, **sim245.s** and **coldstrt.s** files, which implement the **__os_coldstar**t routine that controls the cold start process, also implement the **__os_raminit** routine to transcribe program data segments from ROM to the external data memory. This routine is inlined into the **__os_coldstar**t function and is excluded when building **osboot** for simulators. It can also be suppressed by controlling the definition of the **STANDALONE** manifest on the command-line of the assembler or compiler.

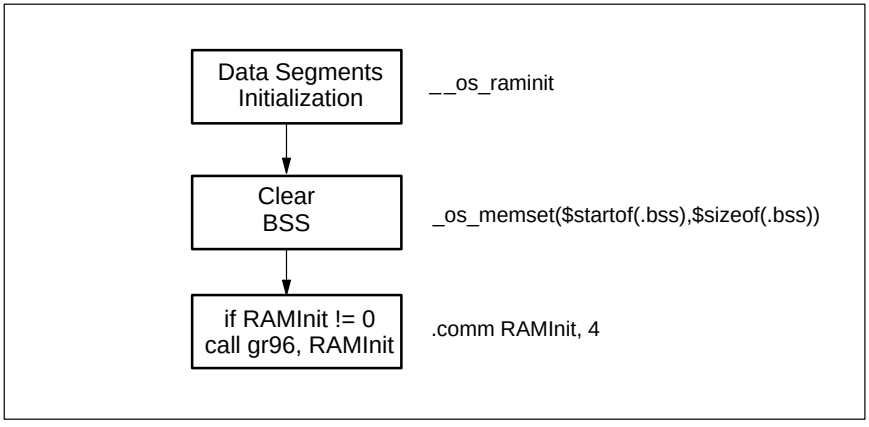

Figure 4-4. Data Segments Initialization

Function

`__os_raminit`

Description

The **__os_raminit** function initializes the external data memory region as necessary. It clears the BSS section and transcribes any initialized data section from ROM to data memory by calling the **RAMInit** function produced by the **romcoff** utility. It defines a BSS symbol, **RAMInit**. At run-time, the content of **RAMInit** is read. If the value read is zero, the variable is in BSS and is initialized to zero. If the value read is non-zero, it assumes that the function **RAMInit** is linked together, and executes it.

The **RAMInit** function generated by the **romcoff** utility requires two stages of linking the application objects with **osboot.** The first link produces a binary image of the application program and **osboot** with text and data sections ordered in memory as they would be during program execution. From this binary image, the **RAMInit** routine is generated using the **romcoff** utility on the sections to be transcribed from ROM space to the external data memory. The **RAMInit** routine generated is then linked with all of the application program objects and **osboot** as the second link phase. This produces a binary image from which the selected sections (e.g., text and lit), can be used to program the EPROMs on the target system.

System Initialization

File

`sysinit.s`

The **sysinit.s** file implements the function that performs system initialization. During system initialization the default trap handlers and defined trap handlers are installed into the vector table. The **__os_targetcfg** global data structure, which stores the target system configuration, is also initialized with the appropriate values.

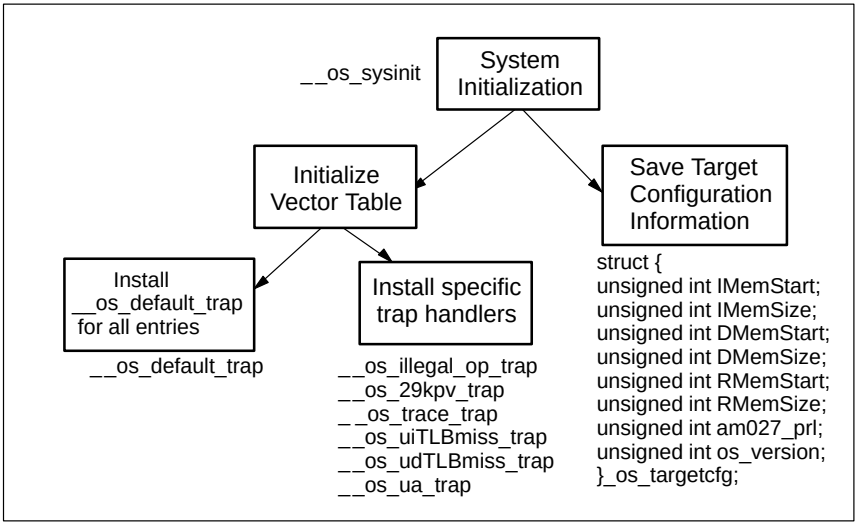

Figure 4-5. System Initialization

Function

`__os_sysinit`

Global Data Variable

`__os_targetcfg`

Link-Time Constants

```
DMemStart
IMemStart
IMemSize
RMemStart
RMemSize
TRAPINROM
```

DMemStart gives the base address of the data memory region.

IMemStart and **IMemSize** are the start address and maximum size of instruction memory on the target system.

RMemStart and **RMemSize** are the start address and maximum size of ROM memory on the target system.

TRAPINROM specifies whether the trap handlers reside in ROM or in instruction memory. A value of 0x2 specifies that the trap handlers reside in ROM. A value of 0x0 specifies that the trap handlers reside in instruction memory. Any other value may cause undefined behavior.

NOTE: Trap handlers should only be located in ROM space (that is, **TRAPINROM** set to 0x2) for those 29K Family microprocessors using 3-bus architecture. For other members of the 29K Family, trap handlers should be located in instruction memory space (**TRAPINROM** set to 0x0).

Description

The **__os_sysinit** function is called during cold-start process to do the following:

- Install the default trap handlers
- Install trap handlers supplied with **osboot**
- Initialize the **__os_targetcfg** data structure according to the target system configuration found

The first argument passed to the **__os_sysinit** function is the total size of data memory found on the target (in register **lr2**).

The default trap vectors for each handler are defined by the **__os_default_trap** function. The value of **TRAPINROM** is OR'ed with the address of the trap handler before it is installed into the vector table.

The specific trap vectors that are installed are the following:

- **__os_illegal_op_trap** as Illegal Opcode trap handler
- **__os_ua_trap** as Unaligned Access trap handler
- **__os_29kpv_trap** as Protection Violation trap handler
- **__os_trace_trap** as Trace trap handler
- **__os_uiTLBmiss_trap** or **__os_uiTLBmiss24x_trap** as User Mode Instruction TLB Miss trap handler
- **__os_udTLBmiss_trap** or **__os_uiTLBmiss24x_trap** as User Mode Data TLB Miss trap handler

The **osboot** data structure **__os_targetcfg** is also initialized with defined link-time constants. The structure of **__os_targetcfg** is:

```
struct {
   unsigned int IMemStart;
   unsigned int IMemSize;
   unsigned int DMemStart;
   unsigned int DMemSize;
   unsigned int RMemStart;
   unsigned int RMemSize;
   unsigned int am027_prl;
   unsigned int os_version;
 } __os_targetcfg;
```

The **IMemStart** and **IMemSize** fields are initialized with values of the link-time constants **IMemStart** and **IMemSize**, respectively. The **DMemStart** field is initialized with the value of the **DMemStart** link-time constant. The field **DMemSize** is initialized with the value passed as incoming argument to **__os_sysini**t (in register **lr2**). The **RMemStart** and **RMemSize** fields are initialized with the values of the link-time constants **RMemStart** and **RMemSize**, respectively. The **am027_prl** field is initialized to -1 (0xffffffff) to indicate that no coprocessor is present (the default). The **os_version** field is initialized with the current version of **osboot**.

Communications Initialization

Files

`eb030hw.s`	for AMD's EB29030 PC plug-in card
`eb29khw.s`	for AMD's EB29K PC plug-in card.
`ez030hw.s`	for AMD's EZ030 stand-alone board
`la030hw.s`	for AMD's Laser29K–030 board
`la200hw.s`	for AMD's Laser29K–200 board
`lcb29khw.s`	for YARC ATM Sprinter PC plug-in card
`pcebhw.s`	for AMD's PCEB29K PC plug-in card
`sa030hw.s`	for AMD's SA29030 stand-alone board
`sa200hw.s`	for AMD's SA29200, SA29205, and SE29240 stand-alone boards
`se040hw.s`	for AMD's SE29040 stand-alone board
`stebhw.s`	for STEP's stand-alone board
`yrev8hw.s`	for YARC Rev 8 PC plug-in card

NOTE: Some of these boards are no longer available commercially, but are still in use.

The ***hw.s** files listed above implement the **__os_initcomm** function that initializes the communications interface for the hardware platforms supported by AMD. The communications interface can be either a shared memory interface (i.e., PC plug-in cards), or serial communications link (i.e., stand-alone boards).

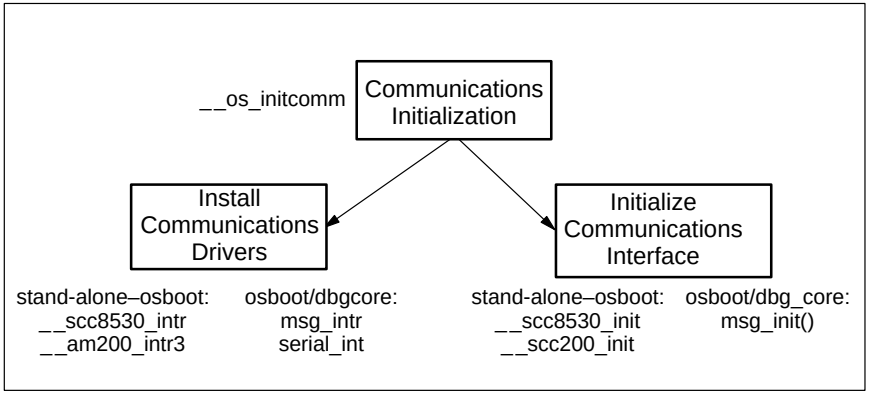

Figure 4-6. Communications Initialization

Function

```
__os_initcomm
```

Associated Functions

For the Stand-Alone OSBOOT/Application Model:

```
__scc8530_init, __scc8530_intr, __scc8530_putchar,
__scc200_init,  __am200_intr3,
__scc200_tx_intr,
__scc200_rx_intr
```

For the OSBOOT/DBG_CORE Model:

```
_msg_init
msg_intr
serial_int
```

Description

The **__os_initcomm** function is called from the **__os_coldstart** function during cold start to initialize the communications interface of the target system.

Stand-Alone OSBOOT/Application Model

For serial-communications interfaces, the **__os_initcomm** function initializes **__putchar_table** with the functions to write a character out of the serial port. It then installs the interrupt vector for the serial device into the vector table and calls the device-specific routine to initialize the device. The serial-communications device functions and their interrupt handlers are defined in the **scc8530.s** and **scc200.s** files.

OSBOOT/DBG_CORE Model

In the OSBOOT/DBG_CORE Model, the **__os_initcomm** function installs the interrupt vector for the **dbg_core** message communications interface in the vector table. The interrupt handlers installed are **msg_intr** for shared memory interface, and **serial_int** for serial-communications interface. The offset into the vector table is defined as a constant in the file corresponding to the target system.

After installing the message interrupt vector, **__os_initcomm** transfers control to the **_msg_init** function inside the **dbg_core** message system. The **_msg_init** function initializes the message system and then returns via **lr0** to the cold-start process.

Chapter 5

OSBOOT Trap Handlers

This chapter describes the following **osboot** trap handlers:

- Protection Violation trap handler on page 5-2
- Unaligned Access trap handler on page 5-4
- Arithmetic trap handlers on page 5-7

Trap handlers can reside either in the ROM space or the instruction memory space of a 29K Family microprocessor. The link-time constant, **TRAPINROM**, specifies where trap handlers will reside. If **TRAPINROM** is set to 0x2, trap handlers are located in ROM space. If **TRAPINROM** is set to 0x0, trap handlers are located in instruction memory space.

NOTE: Trap handlers should only be located in ROM space (that is, **TRAPINROM** set to 0x2) for those 29K Family microprocessors using 3-bus architecture. For other members of the 29K Family, trap handlers should be located in instruction memory space.

Protection Violation Trap Handler

The protection violation trap handler, **__os_29kpv_trap**, is defined in the **tr_pvspr.s** file. The protection violation trap handler is installed on target systems to emulate accesses to special CPU registers in the range gr160–gr164**.** Registers in this range are treated as virtual registers in support of floating-point operations. Table 5-1 lists each of these registers and its contents.

NOTE: On the Am29050 processor, these registers are actually implemented and do not trap. These registers will also be implemented in some future processors.

Table 5-1. Special Virtual Registers

Register	Mnemonic	Description
gr160	fpe	Floating-point environment
gr161	inte	Integer environment
gr162	fps	Floating-point status
gr164	exop	Exception opcode

Access to the virtual registers is provided by trapping **MTSRIM**, **MTSR**, and **MFSR** instructions that reference these registers. The protection violation trap handler receives control when an instruction references an unimplemented register at its entry point. The trap handler performs the following steps:

1. Accesses the **ops** register to determine if the offending instruction is in RAM or ROM. The trap handler accesses the instruction by loading the contents of the address pointed to by the **pc1** register.
2. Determines which of the **MTSRIM**, **MTSR**, or **MFSR** instructions caused the trap. If none of these instructions caused the trap, the handler jumps to the **__os_unexpected_trap** handler.
3. Jumps to the subhandler for each instruction type to decode the value to be stored (in the case of **MTSRIM** and **MTSR** instructions), and the register being referenced. In the case of **MFSR** instructions, only the register being referenced is needed.

4. Calls a unique handler for each case when the register number (and value, if necessary) is determined, depending on whether the instruction intends to move to or from the special register.
5. When the handler returns, it restores the **pc0** and **pc1** registers and returns to the user program.

The unique handlers referenced by **__os_29kpv_trap** are listed in Table 5-2.

Table 5-2. Unique Protection Violation Trap Handlers

Name	Description
EXOP_read	Loads from exception opcode
EXOP_write	Stores to exception opcode
FPE_read	Loads from floating-point environment
FPE_write	Stores to floating-point environment
FPS_read	Loads from floating-point status
FPS_write	Stores to floating-point status
INTE_read	Loads from integer environment
INTE_write	Stores to integer environment

Unaligned Access Trap Handler

The 29K Family of microprocessors supports only byte addressing of information loaded from or stored to memory. In the case of word or half-word accesses, the 29K Family microprocessor either ignores or forces alignment in most cases. The low-order two bits of an address indicate the alignment of the data being accessed. An unaligned access is determined by the value of the OPT bits and the two least significant bits of the address for load and store instructions. Only accesses to instruction or data memory are checked; alignment is ignored for coprocessor transfers. Table 5-3 defines the various OPT bits, and Table 5-4 lists the OPT bit and address bit combinations that result in an unaligned access.

Table 5-3. Option (OPT) Bit Definitions

AS	OPT2	OPT1	OPT0	Definition
x	0	0	0	Word access
x	0	0	1	Byte access
x	0	1	0	Half-word access
0	1	0	0	Instruction ROM access
0	1	0	1	Cache control

The AS bit refers to the address space being referenced. For translated load and store operations, the AS bit must be 0. If the AS bit is 1 for a translated load or store operation, a protection violation occurs.

Table 5-4. Unaligned Access Combinations

OPT2	OPT1	OPT0	A1	A0	Description
0	0	0	1	0	Unaligned word access
0	0	0	0	1	Unaligned word access
0	0	0	1	1	Unaligned word access
0	1	0	0	1	Unaligned half-word
0	1	0	1	1	Unaligned half-word

Whether a trap occurs when an unaligned access is detected depends on the setting of the trap unaligned (TU) bit of the **cps** register. If any of the conditions indicated in Table 5-4 occur, and the TU bit is 1, the unaligned access trap handler is invoked.

The unaligned access trap handler (in the **uatrap.s** file) receives control at the entry point **__os_ua_trap**. The steps taken by this handler are listed below:

1. Accesses the channel control (**chc**) register to determine if the data being accessed is needed and if the **chc** register contents are valid. If the data is not needed, or the contents are invalid, the trap handler returns to the interrupted program; otherwise, the handler continues.
2. Saves the contents of the Old Processor Status **(ops)** register, the channel address (**cha**), channel data (**chd**), and channel control **(chc)** registers to the memory stack.
3. Determines if a multiple operation is in progress (i.e., **LOADM** or **STOREM** instruction). If a multiple operation is in progress, the contents of **pc0** and **pc0+4** are saved; otherwise, the **pc1** and **pc0** registers are saved.
4. Saves the contents of the arithmetic logic unit status (**alu**) and indirect pointer (**ipa**, **ipb**, and **ipc**) registers.
5. Leaves freeze mode by setting the FZ bit in the **cps** register to 0.
6. Sets up the **cps** register to reflect the contents of the **ops** register and the content of the Physical Address (PA) bit from the trapped load or store instruction, and enables all subsequent interrupts and traps (by setting the DI and DA bits of the status to 0). This allows traps to nest, if any occur while the subsequent operations are in progress.
7. Jumps to the subhandler based on status concerning the interrupted instruction: user or supervisor mode; **cfg** register DW bit setting; and **chc** register LS, ML, ST, and LA bits.

The remainder of the **uatrap.s** file contains the unique handlers for each of the possible offending instructions, according to the mode in which it was executed. Table 5-5 lists the names of the unique handlers and the instruction types for which unaligned accesses are resolved.

When each of the unique handlers completes the load or store operation, a common label, **restore,** receives control. This section of code restores the contents of the registers saved on entry to the handler, and sets up the **ops** register prior to returning to the interrupted program.

Table 5-5. Unaligned Access Unique Handlers

Name	Instruction Type
load_u	User-mode load
store_u	User-mode store
loadl_u	User-mode load and lock
storel_u	User-mode store and lock
load_s	Supervisor-mode load
store_s	Supervisor-mode store
loadl_s	Supervisor-mode load and lock
storel_s	Supervisor-mode store and lock
loadm_u	User-mode load multiple
storem_u	User-mode store multiple
loadm_s	Supervisor-mode load multiple
storem_s	Supervisor-mode store multiple
load_hu	User-mode load half-word, DW=0
store_hu	User-mode store half-word, DW=0
loadl_hu	User-mode load and lock half-word, DW=0
storel_hu	User-mode store and lock half-word, DW=0
load_hs	Supervisor-mode load half-word, DW=0
store_hs	Supervisor-mode store half-word, DW=0
loadl_hs	Supervisor-mode load and lock half-word, DW=0
storel_hs	Supervisor-mode store and lock half-word, DW=0
load_hudw	User-mode load half-word, DW=1
store_hudw	User-mode store half-word, DW=1
loadl_hudw	User-mode load and lock half-word, DW=1
storel_hudw	User-mode store and lock half-word, DW=1
load_hsdw	Supervisor-mode load half-word, DW=1
store_hsdw	Supervisor-mode store half-word, DW=1
loadl_hsdw	Supervisor-mode load and lock half-word, DW=1
storel_hsdw	Supervisor-mode store and lock half-word, DW=1

Arithmetic Trap Handlers

The arithmetic trap handlers provided with **osboot** perform floating-point and long-integer arithmetic operations for the 29K Family of processors. The arithmetic trap handlers emulate IEEE floating-point operations, including IEEE integer operations. These trap handlers are not designed for optimum floating-point performance. Instead, they provide complete object-code compatibility with future generations of 29K Family processors. For optimum floating-point performance, it is necessary to code in-line accelerator instructions.

The integration of the floating-point trap handlers with an operating system raises a number of key issues. The options available to the system integrator may drastically affect the system performance, although no simple formula or well-defined set of criteria exists.

The customizations required are limited to the following macros:

```
enter_trap_routine
exit_trap_routine
```

These macros perform environment control and register save/restore on entry and/or exit to the trap handlers. The following notes are intended for those familiar with system calling conventions and the 29K Family of microprocessors at the register transfer level.

- Decide whether freeze mode should be turned off or on for an emulation trap. The source code, as provided, turns off freeze mode.
- Typical operation may take 100 cycles or more. If the user's system can tolerate a few microseconds of frozen state, the entire operation can be done with the freeze mode on. A freeze-mode handler is much simpler, a bit faster (saves at least 10 cycles), and requires fewer dedicated registers.
- Decide which registers can be used by the arithmetic trap handlers. The source code, as provided, uses a maximum number of registers for optimum performance. However, if not enough registers are available, some registers will have to be saved on the memory stack on entry to and restored on exit from each trap handler.

NOTE: The registers gr64–gr95 are reserved for the operating system and normally are protected. These registers are not protected in systems that allow arbitrary floating-point programs to run in supervisor mode. Therefore, since protection of the registers is not available, code must be added to each floating-point handler to ensure registers gr64–gr95 are not used as arguments. Make sure that none of these registers are used elsewhere in the operating system. If the traps are running with freeze off and interrupt on, ensure that none of the floating-point registers overlap with registers used in any other interrupt handlers.

- Once registers are allocated, define their assignments in the **register.ah** file. If all trap handlers are to be run with freeze mode on, there is no need to save the **pc0**, **pc1**, and **ops** registers.
- The information necessary to write the **enter_trap_routine**, **exit_trap_routine**, **enter_29027_trap_routine**, and **exit_29027_trap_routine** macros is located in the **tr_decl.h** file.

Changes must be made to the **enter_trap_routine** and **exit_trap_routine** macros, depending on the system choices made. The sections below describe the requirements for each choice.

Freeze on and all registers available

These macros are empty. The macros are shipped in this form; **osboot** files allow a sufficient number of working registers.

Freeze on and not all registers free

Allocate an area of physical memory in which to save registers in the **enter_trap_routine** macro and to restore the registers in the **exit_trap_routine** macro.

Freeze off and all registers free

In the **enter_trap_routine** macro, save the **ops**, **pc1**, and **pc0** registers and turn off the freeze mode. In the **exit_trap_routine** macro, first turn on freeze mode and then restore the **ops**, **pc1**, and **pc0** registers.

Freeze off, not all registers free, and physical stack

In the **enter_trap_routine** macro, save the **ops**, **pc1**, and **pc0** registers (and any other required registers) on the physical stack, then turn off the freeze mode. In the **exit_trap_routine** macro, first turn on freeze mode and then restore all saved registers. Remember that **LOADM/STOREM** instructions cannot be used in freeze mode. This is the likely case for embedded systems where everything runs in physical memory.

Freeze off, not all registers free, and virtual stack

In the **enter_trap_routine** macro, save **ops**, **pc1**, and **pc0** in either registers or physical memory. Turn on freeze mode and virtual data mapping, but leave interrupts off. Next, switch to the virtual stack, then save the copies of **ops**, **pc1**, **pc0**, and the other registers. Finally, turn on the interrupts. Do the inverse operations in the reverse order in the **exit_trap_routine** macro. This is the likely case for systems such as UNIX in which a number of user processes run in virtual memory.

When running UNIX or a time-sharing system with shared libraries (in which the libraries' data pages are copy-on-write), the floating-point handlers can be implemented efficiently in user mode, similar to the way the spill/fill traps are implemented.

In this case, the floating-point routines cannot be passed invalid arguments, and the floating-point registers can be in static memory data areas (with respect to the libraries).

This approach may cause more load and store instructions than the other strategies, but there is no need to save and restore the floating-point state across a context switch.

Chapter 6

HIF Run-Time Services

When **osboot** is used with simulators or in stand-alone applications, the HIF kernel is invoked immediately after completion of the cold-start process. In the OSBOOT/DBG_CORE Model, the **osboot** temporarily transfers control to **dbg_core** by calling the **dbg_control()** function of **dbg_core**. When the **dbg_control()** function returns control to **osboot**, the HIF kernel is invoked. The HIF kernel start-up function is **__hif_startup**.

The **__hif_startup** function starts up the HIF kernel by initializing its data structures, and prepares an execution environment for the user application program. The execution from the start up of the HIF kernel to the beginning of the execution of the user application program is the warm-start process.

This chapter describes the functions provided by the **osboot** HIF kernel and their associated files.

HIF Kernel Start-Up Module

Files

`hif.s, hifvect.c`

The **hif.s** file implements the start-up program for the HIF kernel that is called after the completion of the cold start process. It initializes the HIF services and creates a run-time environment for the application program.

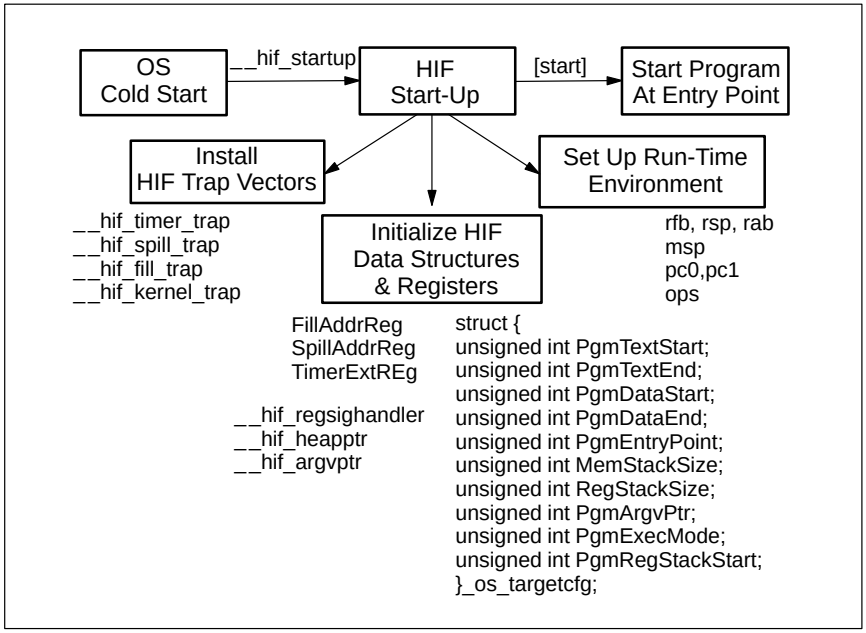

Figure 6-1. HIF Kernel Start-Up Module

Function

`__hif_startup`

Associated Function

`_hif_vectinit(long *VAB, int trapinrom)`

Link-Time Constant

`TRAPINROM`

TRAPINROM indicates whether the trap handlers reside in ROM space or in instruction memory space. A **TRAPINROM** setting of 0x2 (two) indicates that the trap handlers reside in ROM space. A **TRAPINROM** setting of 0x0 indicates that the trap handlers reside in instruction memory space.

NOTE: Trap handlers should only be located in ROM space (that is, **TRAPINROM** set to 0x2) for those 29K Family microprocessors using 3-bus architecture. For other members of the 29K Family, trap handlers should be located in instruction memory space (**TRAPINROM** set to 0x0).

Description

The **__hif_startup** function is invoked at the end of the cold start process. This function calls **__hif_vectinit** to install the HIF vectors for Timer trap, Spill trap, Fill trap, and the HIF kernel trap into the vector table. The vectors installed are:

__hif_timer_trap	for Timer trap (0xE)
__hif_spill_trap	for Spill trap (0x40)
__hif_fill_trap	for Fill trap (0x41)
__hif_kernel_trap	for HIF Kernel trap (0x45)

After installing the vectors, the **__hif_startup** function initializes the **SpillAddrReg** and **FillAddrReg** kernel static registers defined in the **register.s** file to the **__hif_spillhandler** and **__hif_fillhandler** routines, respectively. It also initializes the TMR and TMC special purpose registers and the **TimerExt** kernel static register before enabling the timer.

Next, **__hif_startup** initializes the data structures used by the HIF services, including the program information contained in the **__hif_pgminfo** structure. The **__hif_pgminfo** structure is as follows:

```
struct {
  unsigned int    PgmTextStart;
  unsigned int    PgmTextEnd;
  unsigned int    PgmDataStart;
  unsigned int    PgmDataEnd;
  unsigned int    PgmEntryPoint;
  unsigned int    PgmMemStackSize;
  unsigned int    PgmRegStackSize;
  unsigned int    PgmArgvPtr;
  unsigned int    PgmExecMode;
  unsigned int    PgmRegStackStartAddr;
} __hif_pgminfo;
```

Program information is provided by the external loader or hard coded in the software at compile time. The **stdalone.ah** file defines the constants used by the software (when there is no external loader) and the **_os_initpgminfo** macro, which initializes the **__hif_pgminfo** structure at HIF start-up time.

The data structures used by the HIF services are:

```
unsigned int _hif_heapptr;      /* Holds the heap base address */
unsigned int _hif_argvptr;      /* Holds argv pointer address */
unsigned int _hif_regsighandler;/* Holds the registered signal */
                                /* handler address */
```

Using the target system configuration information in the **__os_targetcfg** structure and the application program information in the **__hif_pgminfo** data structure, this function sets up the run-time environment for the program, including:

- Setting up the register and memory stacks at the top of the memory (high memory)
- Setting up the OPS register for the application program
- Flushing TLBs
- Setting up the PC0 and PC1 registers to the program's entry point
- Performing an IRETINV instruction to start program execution

Run-Time Environment

The HIF kernel requires the information necessary to set up the run-time environment to reside in the **__hif_pgminfo** and the **__os_targetcfg** data structures. The **__os_targetcfg** data structure is initialized by the **osboot** bootstrap module during cold start according to the target system configuration. The **__hif_pgminfo** data structure is either initialized by the software with predefined values or provided by the external loader that downloaded the program. The external loader is required to provide the information to initialize the **__hif_pgminfo** structure in registers gr96 through gr105.

Table 6-1 lists the registers used by the external loader to provide the program information to the HIF kernel, and the symbolic names and addresses used by the HIF kernel when no external loader is present. Table 6-1 explains run-time setup data.

Table 6-1. Registers and Symbolic Names for Run-Time Information

Run-Time Setup Data	External Loader	No External Loader
User text start address	gr96	PgmTextStart
User text end address	gr97	etext
User data start address	gr98	DMemStart
User data end address or heap base	gr99	Determined at run-time
Application program's entry point	gr100	Start
Memory stack size	gr101	PgmMemStackSize
Register stack size	gr102	PgmRegStackSize
Argv start address	gr103	Determined at run-time
Program Execution mode	gr104	PgmExecMode
Register Stack Start Address	gr105	PgmRegStackStart

Table 6-2. Run-Time Setup Data

Synopsis	Definition
User Text Start Address	Contains the lowest address of the text region of the object. Not used in setting up run-time environment. Used by TLB handlers when running in protected mode.
User Text End Address	Contains the highest address of the text region. Not used in setting up run-time environment. Used by TLB handlers when running in protected mode.
User Data Start Address	Contains the lowest address of the data region (all nontext region). Not used in setting up run-time environment. Used by TLB handlers when running in protected mode.
User Data End Address/ Base	Contains the highest address of the Heap data region (all non-text region) and the first address of the heap. Therefore, program arguments (**argv**) are included as part of the data region. Used during setup of the run-time environment. Used to initialize **__hif_heapptr**, which stores the heap base address.
Application Program's Entry Point	Entry point of the program loaded. Used to initialize the program counters (**PC0**, **PC1**) when setting up the run-time environment. Control is then transferred to the user program at the entry point by **iret**.
Memory Stack Size	Contains the memory stack size.
Register Stack Size	Contains the register stack size.
Argv Start Address	Contains the starting address of **argv**. Used to initialize **__hif_argvptr**, which stores the pointer to **argv**.

Synopsis	Definition	
Program Execution Mode	Is a 32-bit value to specify the execution mode for the application program. The following bits are currently defined:	
	Bit 31 set	User mode—no translation
	Bit 30 set	User mode—translate data
	Bit 29 set	User mode—translate instruction
	Bit 28 set	Supervisor mode—no translation
	None set	User mode—translate instruction and data
Register Stack Start Address	Gives the highest memory location available. This must be set to either 0 or to the highest memory location by the loader or by the simulator. The register stack is set up at the top of memory, which is also the register stack start address.	

After the run-time environment is set up, the control of the processor is transferred to the application program at its entry point. At that time, the global registers gr96 and gr97 are initialized with coprocessor mode information. Local registers lr0 and lr1 are initialized to execute an **exit( )** HIF system call if the application program immediately returns control back to **osboot.**

Register Stack and Memory Stack Arrangement

The memory stack resides directly below the register stack at the top of the data memory (high memory).The register stack resides *VARARG_SPACE* bytes below the highest addressable data memory location, and the register stack pointers (**rsp**, **rfb**, and **rab**) are initialized. Both stacks grow from high to low memory locations.

Figure 6-2 below shows the upper portion of the data memory space.

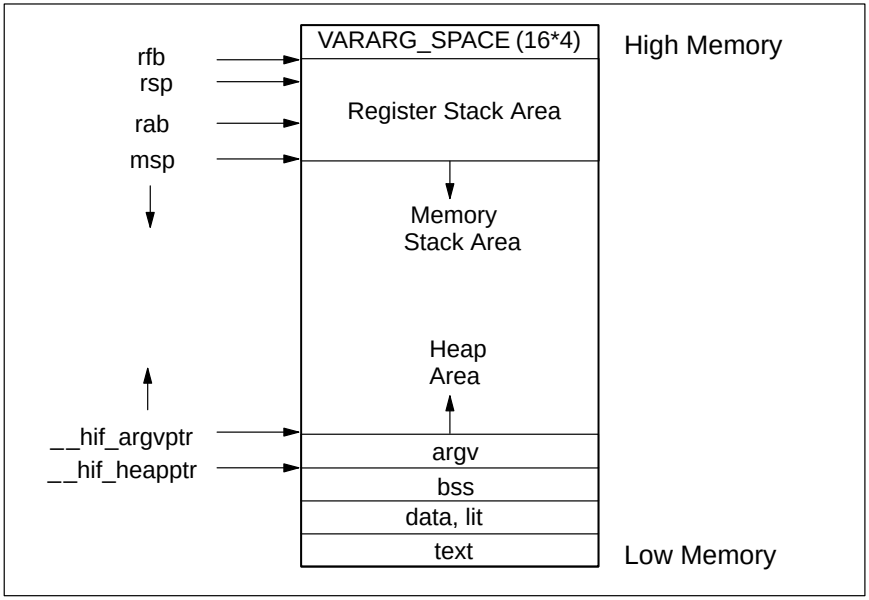

Figure 6-2. Register Stack and Memory Stack Arrangement

HIF Services

File

`hifserv.s`

The **hifserv.s** file implements the HIF trap routines for the Timer trap, Spill trap, Fill trap, and the HIF kernel trap.

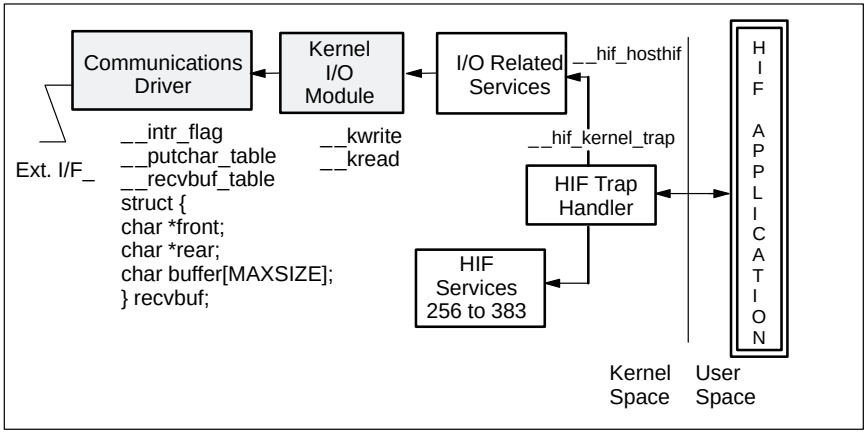

Figure 6-3. HIF Services Module

Functions

```
__hif_timer_trap
__hif_spill_trap
__hif_fill_trap
__hif_kernel_trap
```

Associated Function

`__hif_hosthif`

Link-Time Constants

```
TicksPerMillisecond
ClockFrequency
```

TicksPerMillisecond and **ClockFrequency** are defined in the **.lnk** linker command file, and specify the target processor's clock frequency. These values are used to implement certain operating-system services.

Description

__hif_timer_trap implements the HIF kernel's timer interrupt handler. It resets the interrupt and increments the Timer extension kernel static register, **TimerExt,** by 1, and returns from the interrupt.

__hif_spill_trap implements the register spill trap. It sets the PC0 and PC1 registers with the address contained in the **SpillAddrReg** kernel static register and returns from the interrupt. The contents of the **SpillAddrReg** are initialized at HIF start-up time and can also be initialized using a HIF **setvec** service.

__hif_fill_trap implements the register fill trap. It sets the PC0 and PC1 registers with the address contained in the **FillAddrReg** kernel static register and returns from the interrupt. The contents of the **FillAddrReg** are initialized at HIF start-up time and can also be initialized using a HIF **setvec** service.

__hif_kernel_trap implements the HIF kernel services that are invoked by the HIF kernel trap 0x45. (The services provided are documented in the *Host Interface (HIF) Specification* document.) HIF service numbers below 256 are implemented by a separate routine, **__hif_hosthif,** which may use the services of a remote intelligent host to perform the operations. HIF services numbered between 256 and 383 are implemented in this **__hif_kernel_trap** routine.

Host HIF Services

File

`hosthif.s, msgio.s, simtraps.s`

The **hosthif.s** file implements the **__hif_hostif** routine for the Stand-Alone OSBOOT/Application Model. The **simtraps.s** file implements the **__hif_hosthif** routine for the OSBOOT/Simulator Model. The **msgio.s** file implements the **__hif_hosthif** routine for the OSBOOT/DBG_CORE Model.

Function

`__hif_hosthif`

Stand-Alone OSBOOT/Application Model

In the Stand-Alone OSBOOT/Application Model, where the application is linked with **osboot** and programmed into EPROMs, the host HIF services are handled by the kernel I/O module of **osboot** (**kio.s**) or by the functions provided by the application program itself.

OSBOOT/Simulator Model

The simulator intercepts all the host HIF service requests made by the application program and processes them using its host operating system. It writes the results back to the simulated memory and registers of the 29K Family target program and resumes execution of the program.

OSBOOT/DBG_CORE Model and Stand-Alone OSBOOT/DBG_CORE/Application Model

This section describes the OSBOOT/DBG_CORE Model or the stand-alone OSBOOT/DBG_CORE/Application Model. The information in this section applies to either model equally. In the OSBOOT/DBG_CORE Model shown in Figure 6-4 , some of the HIF kernel services are provided with the help of an external HIF support module in **montip**. The **montip** runs on an intelligent host computer system and communicates with the **dbg_core**'s message system. It receives requests from the HIF kernel, services them using the intelligent host operating system, and sends the results back to the HIF kernel. The HIF kernel uses the message system of the **dbg_core** to implement the host HIF services from 0 through 255.

When the application program running in user mode issues a HIF system call, a HIF kernel trap (0x45) is taken. The HIF kernel trap handler,**__hif_kernel_trap**, determines from the service number contained in gr121 if it requires the services of the external HIF support module in **montip**. The service requests that require the support of **montip** are processed by the **__hif_hosthif** function, defined in the **msgio.s** file.

At the entry point of the **__hif_hosthif** function:

- Global register gr121 contains the HIF service number.
- Local registers lr2 through lr4 contain HIF service arguments.

Based on the requested service, the results are returned in register gr96 (and gr97 if necessary), and the error code is returned in gr121. A logical TRUE (0x80000000) is returned in gr121 if the service was completed successfully. Otherwise, one of the defined HIF error codes is returned in gr121. The program making the system call must check the value returned in gr121 before using the results.

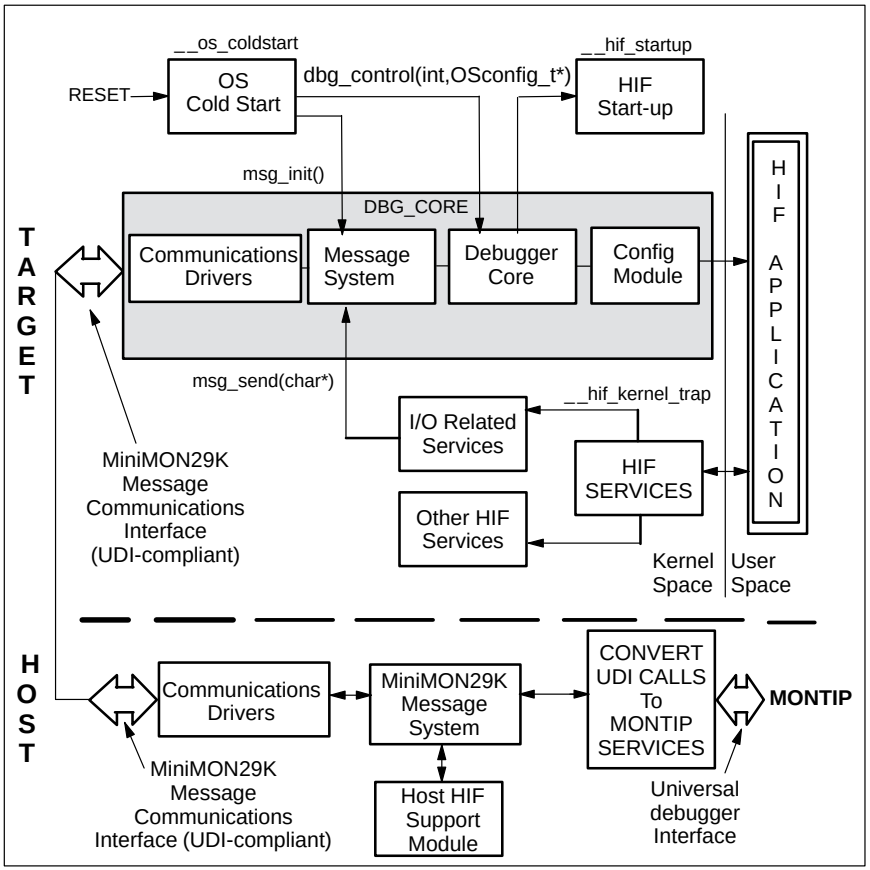

Figure 6-4. OSBOOT/DBG_CORE Model and MONTIP

DBG_CORE Message System Interface

The **__hif_hosthif** function of the HIF kernel communicates with the external HIF module in **montip,** which is running on an intelligent host computer system, using MiniMON29K messages. The message structure and semantics are defined in the *Target Interface Process: MONTIP* manual. The messages are transmitted and received with the help of the services of the MiniMON29K **dbg_core** message system. The **dbg_core** message system provides the following interface:

msg_rbuf

msg_rbuf is the receive buffer of the **dbg_core** message system. Incoming messages from **montip** are received by the communications interface and placed into this buffer. The message system is notified when a valid message is received.

msg_send(msg_t *)

The **msg_send()** function can be used to send a MiniMON29K message to **montip**. It takes a pointer to the outgoing message as its first argument. It returns a value of 0 (zero) in gr96 if the message was successfully sent, and a -1 (0xffffffff) if the message communications channel is busy with a previous request.

msg_wait_for()

The **msg_wait_for()** function can be used to receive an incoming message. This function waits and returns when a valid message has been received in the message system's receive buffer (**msg_rbuf**). A -1 (0xffffffff) is returned to indicate that the receive buffer has a valid message. However, when operating in interrupt mode, it returns 0 (zero) immediately. When the message communications interface is interrupt-driven, the kernel is interrupted when a new message arrives.

The HIF kernel provides the following entry point for the message system:

```
os_V_msg
```

When a new message arrives, the message system examines the message code and interrupts the kernel if it was an OS message. The message system interrupts by transferring control to the **os_V_msg** label inside the HIF kernel.

How the MiniMON29K Messages are Used

The UDI-compliant MiniMON29K messages used by the **osboot** HIF kernel to communicate with **montip** are described on the pages that follow.

channel1 and channel1_ack message pair

```
struct channel1_msg_t {
   INT32     code;          /* 98 */
   INT32     length;        /* number of bytes to   */
                            /* follow               */
   BYTE      data;          /* 1st data byte of the */
                            /* message              */
};

struct channel1_ack_msg_t {
   INT32     code;           /*  66  */
   INT32     length;         /* number of bytes to  */
                             /* follow (equals 4)    */
   INT32     nbytes_written; /* num of bytes written */
};
```

A **channel1** message (code 98) is sent by the HIF kernel to **montip** when an application program makes a HIF system call to write to the standard output device. The message includes the bytes to be printed on the standard output device.

The kernel then waits for a **channel1_ack** message response from **montip** before resuming the application program. Global register gr96 is set with the value returned in **channel1_ack** by **montip** as the number of bytes successfully written (the **nbytes_written** field), and global register gr121 is set to logical TRUE if the message was successfully sent to **montip**.

channel2 and channel2_ack message pair

```
struct channel2_msg_t {
   INT32     code;          /* 99 */
   INT32     length;        /* number of bytes to   */
                            /* follow               */
   BYTE      data;          /* 1st data byte of the */
                            /* message              */
};

struct channel2_ack_msg_t {
   INT32     code;          /* 67  */
   INT32     length;        /* number of bytes to   */
                            /* follow (equals 4)    */
   INT32     nbytes_written;/* number of bytes     */
                             /* written              */
};
```

A **channel2** message (code 99) is sent by the HIF kernel to **montip** when the application program makes a HIF system call to write to the standard error device. The message includes the bytes to be printed on the standard error device. The kernel then waits for the **channel2_ack** message response from **montip** before resuming the application program. Global register gr96 is set with the value returned in **channel2_ack** by **montip** as the number of bytes successfully written (the **nbytes_written** field), and global register gr121 is set to logical TRUE if the message was successfully sent to **montip**.

stdin needed and stdin needed ack message pair

```
struct stdin_needed_msg_t {
    INT32     code;          /* 100  */
    INT32     length;        /* number of bytes to   */
                             /* follow (equals 4)     */
    INT32     nbytes_needed; /* maximum num.of bytes */
                             /* needed                */
};
struct stdin_needed_ack_msg_t {
    INT32     code;          /* 68  */
    INT32     length;        /* number of bytes to   */
                             /* follow               */
    BYTE      data;          /* 1st data byte of the */
                             /* message              */
};
```

A **stdin needed** message (code 100) is sent by the HIF kernel to **montip** when the application program makes a HIF system call to read from the standard input device. This message is sent only when the input mode is blocking and synchronous. The default input mode of the HIF kernel is blocking and synchronous. The application program can change the input mode using a HIF **ioctl** system call.

In synchronous input mode, the HIF kernel sends a **stdin needed** message to **montip** requesting data from the standard input device. It then waits for a **stdin needed ack** message response from **montip**, which includes the input data from the standard input device. The kernel copies the input data into the application program's input buffer. Global register gr96 is set to the number of input bytes received, and global register gr121 is set to logical TRUE if the message was successfully sent to **montip**.

When operating in asynchronous mode, the HIF kernel and **montip** use the **channel0** and **channel0_ack** message pair to send characters, possibly interrupting the running application program. The characters received by the kernel are returned to the application program when it makes a HIF system call to read from standard input.

channel0 and channel0_ack message pair

```
struct channel0_msg_t {
   INT32      code;          /* 65  */
   INT32      length;        /* number of bytes to */
                             /* follow (equals 1)  */
   BYTE       data;          /* the input byte     */
};

struct channel0_ack_msg_t {
   INT32      code;          /* 97  */
   INT32      length;        /* number of bytes to */
                             /* follow (equals 0)  */
};
```

During asynchronous input mode, **montip** sends the characters received from the standard input device to the HIF kernel using the **channel0** message. A **channel0** message is used to send one input byte. The **dbg_core** message system interrupts the HIF kernel when a **channel0** message is received. The HIF kernel stores the input data byte and saves it in a standard input device buffer. The HIF kernel sends a **channel0_ack** message to **montip** to acknowledge the receipt, and resumes the execution of the interrupted application program.

stdin mode and stdin mode ack message pair

```
struct stdin_mode_msg_t {
   INT32      code;          /* 101  */
   INT32      length;        /* number of bytes to    */
                             /* follow (equals 4)     */
   INT32      input_mode;    /* requested input mode  */
};

struct stdin_mode_ack_msg_t {
   INT32      code;          /* 69 */
   INT32      length;        /* number of bytes to   */
                             /* follow               */
   INT32      previous_mode; /* returns the previous */
                             /* input mode           */
};
```

The HIF kernel sends the **stdin mode** message (code 101) to **montip** to specify the input mode or a change in the input mode for the standard input device. The requested input mode is coded into the **input_mode** field of the message. It then waits for a **stdin mode ack** message response from **montip,** which includes the previous mode of the standard input device. The HIF kernel saves the mode values and resumes the execution of the application program. A **stdin mode** message is used when the application program issues a HIF **ioctl** system call to change the standard input mode.

HIF call and HIF call return message pair

```
struct hif_call_msg_t {
   INT32      code;          /* 96 */
   INT32      length;        /* number of bytes to    */
                             /* follow (equals 16)    */
   INT32      service_number;/* HIF service number    */
   INT32      lr2;           /* HIF service 1st input */
                             /* argument in lr2       */
   INT32      lr3;           /* HIF service 2nd input */
                             /* argument in lr3       */
   INT32      lr4;           /* HIF service third     */
                             /* input argument in lr4 */
};

struct hif_call_rtn_msg_t {
   INT32      code;          /* 64 */
   INT32      length;        /* number of bytes to    */
                             /* follow (equals 16)    */
   INT32      service_number;/* HIF service number    */
   INT32      gr121;         /* HIF service completion*/
                             /*code(TRUE or errno)    */
   INT32      gr96;          /* HIF service 1st return*/
                             /* value                 */
   INT32      gr97;          /* HIF service second    */
                             /* return value          */
};
```

When the application program issues a HIF system call to perform an I/O operation on the file system of the host computer running **montip**, the HIF kernel sends a **HIF call** message (code 96) to **montip**. The message includes the HIF service number and up to three input arguments. The HIF services currently defined do not take more than three input arguments. The kernel then waits for a **HIF call return** message response from **montip** before resuming the application program. Depending on the HIF service requested, **montip** may invoke the **dbg_core**, which interrupts the kernel waiting for the **HIF call return** message. **montip** sends a GO message to switch the context from **dbg_core** to the interrupted kernel. It then sends the results of the HIF service in the **HIF call return** message. The kernel sets global register gr121 to the completion code sent by **montip**. Depending on the HIF service requested (determined from the **service_number** field), the kernel sets registers gr96 and gr97 with the values returned by **montip**.

Implementation of os_V_msg Message Interrupt Handler

When the **dbg_core** message system receives a message for the HIF kernel, it interrupts the kernel by transferring control at the **os_V_msg** label. **os_V_msg** is a *virtual* interrupt vector and is defined in the **msgio.s** file. The following code sample shows the **os_V_msg** interrupt handler:

```
os_V_msg:
     pushreg   gr96, msg                  ; PUSH gr96
     ; check for Channel0 message (asynchronous)
     const     gr96, _msg_rbuf
     consth    gr96, _msg_rbuf
     load      0,0, gr96, gr96            ; get message code
     cpeq      gr96, gr96,CHANNEL0_MSGCODE
     jmpt      gr96, process_channel0_msg ; restores
     nop                                  ; gr96 and msp

     ; not a channel0 message (synchronous message)
     const     gr96, __hif_msgwaiting     ; set the flag
                                          ; polled
     consth    gr96, __hif_msgwaiting     ; by the kernel
                                          ; while waiting.
     store     0, 0, gr96, gr96           ; Set to non-zero
                                          ; value
     popreg    gr96, msp                  ; POP gr96
     iret
```

The HIF kernel waits by polling a flag (memory location), **__hif_msgwaiting**. It loops and waits as long as the flag is set to zero. When a message is received, this flag is set to a non-zero value. The kernel exits the wait loop and processes the incoming message. As the message systems of **dbg_core** and **montip** communicate using synchronous message pairs, the kernel assumes the new message received to be in response for its request and returns to the application program.

Implementation of Message Communications in HIF Kernel

The HIF kernel uses macros to build and send UDI-compliant MiniMON29K messages using the **dbg_core** message system. The **BuildHIFCallMsg** macro shown in the code sample below builds a **HIF Call** message (code 96) in the buffer, **bufaddr**.

```
.macro   BuildHIFCallMsg, bufaddr, tmp1, tmp2
const    tmp1, bufaddr
consth   tmp1, bufaddr
const    tmp2, HIF_CALL_MSGCODE
store    0, 0, tmp2, tmp1        ; msg code
add      tmp1, tmp1, 4
const    tmp2, HIF_CALL_MSGLEN
store    0, 0, tmp2, tmp1        ; msg len
add      tmp1, tmp1, 4
store    0, 0, gr121, tmp1       ; service number
add      tmp1, tmp1, 4
store    0, 0, lr2, tmp1         ; lr2
add      tmp1, tmp1, 4
store    0, 0, lr3, tmp1         ; lr3
add      tmp1, tmp1, 4
store    0, 0, lr4, tmp1         ; lr4
.endm
```

The input parameters **tmp1** and **tmp2** to the macro are the temporary registers to use to build the message. The HIF kernel uses the macro to build a **HIF Call** message in its message buffer, **__hif_msgbuf**, as follows:

```
BuildHIFCallMsg __hif_msgbuf,gr96,gr97
```

The kernel saves global registers gr96 and gr97 before invoking the macro.

Another macro, **SendMessageToMontip**, is defined to send the message built in **__hif_msgbuf** to **montip**. The code below shows the definition of the **SendMessageToMontip** macro.

```
     .macro SendMessageToMontip, bufaddr
      const       lr2, bufaddr
$1:
      call     lr0, _msg_send   ; function of dbg_core
                                ; message system
      consth   lr2, bufadd
      cpeq     gr96, gr96, 0    ; successfully sent?
      jmpf     gr96, $1         ; no, try again.
      const    lr2, bufaddr
     .endm
```

Before invoking the macro to send a message that has been built with **BuildHIFCallMsg**, the local registers lr0, lr1, and lr2 must be saved, and restored later.

The example below illustrates the implementation of a HIF service routine that uses a **HIF Call** message.

```
pushreg gr96, msp          ; PUSH gr96
pushreg gr97, msp          ; PUSH gr96

BuildHIFCallMsg  __hif_msgbuf, gr96, gr97

pushreg lr0, msp           ; PUSH lr0
pushreg lr1, msp           ; PUSH lr1
pushreg lr2, msp           ; PUSH lr2

SendMessageToMontip __hif_msgbuf

popreg  lr2, msp           ; POP lr2
popreg  lr1, msp           ; POP lr1
popreg  lr0, msp           ; POP lr0

jmp    _wait_for_ack       ; jump to the wait routine
nop
```

As shown, a **HIF Call** message is built in the HIF message buffer, **__hif_msgbuf**, and sent to **montip** using the **dbg_core** message system. After successfully sending the message, it waits for the response message from **montip** in the **_wait_for_ack** routine. The code sample on the following page shows the **_wait_for_ack** routine, which waits for a response from **montip**.

```
_wait_for_ack:
     SaveFZState      gr96, gr97
     mtsrim    chc, 0
     pushreg   lr0, msp              ; PUSH lr0
     pushreg   lr1, msp              ; PUSH lr1
     const     gr96, _msg_wait_for   ; returns immediately
                                     ; interrupt mode
     consth    gr96, _msg_wait_for   ; waits for message,
                                     ; polled mode
     calli     lr0, gr96
     nop

     popreg    lr1, msp              ; POP lr1
     popreg    lr0, msp              ; POP lr0

     jmpt      gr96, process_msg     ; did _msg_wait_for
      nop                            ; return a message


; enable interrupts
     mfsr      gr96, CPS
     andn      gr96, gr96, (DI|DA)
     mtsr      CPS, gr96

     ; wait for a message, poll flag
     const     gr96, __hif_msgwaiting
$6:
     consth    gr96, __hif_msgwaiting
     load      0, 0, gr96, gr96      ; read flag
     cpeq      gr96, gr96, 0         ; compare with zero
     jmpt      gr96, $6              ; yes, no message, loop
     const     gr96, __hif_msgwaiting


     ; message received
process_msg:
     ; clear __hif_msgwaiting flag
     const   gr96, __hif_msgwaiting
     consth  gr96, __hif_msgwaiting
     const   gr97, 0
     store   0, 0, gr97, gr96        ; clear flag

     ; code to process message follows
     ; the handlers restore Freeze state, gr96, gr97
     ; and memory stack
```

The **_wait_for_ack** routine first calls the message system's **msg_wait_for()** function to receive a message. In polled mode, the **msg_wait_for** function waits until a valid message is received in the buffer, **msg_rbuf**, and returns a -1 in gr96. In interrupt mode, the **msg_wait_for()** function returns immediately with gr96 set to 0 (zero). The return value from **msg_wait_for()** is checked by the **_wait_for_ack** routine. If a -1 (0xffffffff) is returned, then that indicates a valid message in the receive buffer, and the **_wait_for_ack** routine calls **process_msg** to process the message received.

If a 0 (zero) is returned, then that indicates that the message interface interrupt driver, and **_wait_for_ack** routine waits for a message interrupt. This is done by polling the **__hif_msgwaiting** flag. When a message interrupt occurs, the **_wait_for_ack** routine stops polling the flag and enters the **process_msg** routine.

The **process_msg** routine processes the incoming message by extracting the information sent by **montip** into gr96 and gr97, then restores the freeze mode state earlier saved by the **_wait_for_ack** routine, and repairs the memory stack altered earlier. The kernel then resumes the execution of the application program.

Chapter 7

Building OSBOOT or OSBOOT/DBG_CORE

AMD's **osboot** software contains the bootstrap code for the 29K Family of microprocessors, the HIF kernel, and the instruction emulation routines for certain arithmetic instructions of the 29K Family that may cause a trap.

The architectural simulator (**sim29**) and the instruction set simulator (**isstip**) use **osboot** as the ROM bootstrap code. The following table gives the names of the **osboot** binary files and their intended target processor. The simulators find the appropriate **osboot** to use based upon the command-line option that specifies which processor to simulate.

Table 7-1. OSBOOT for Simulators

Filename	Target Processor
osb00x	Am29000 and Am29005
osb03x	Am29030 and Am29035
osb040	Am29040
osb050	Am29050
osb20x	Am29200 and Am29205
osb24x	Am29240 and Am29243
osb245	Am29245

osboot is also ported to operate with the MiniMON29K debugger core, **dbg_core**. The linked objects of OSBOOT/DBG_CORE can be built for different target platforms, and can be programmed into EPROMs to provide an environment to develop, debug, and execute embedded application programs.

The batch files for MS-DOS systems and make files for UNIX systems to build **osboot** and **dbg_core** are kept in the **osboot** directory.

MS-DOS Batch Files

`makeosb.bat`	To build **osboot**
`makemon.bat`	To build **osboot/dbg_core**

UNIX Make Files

`makefile.os`	To build **osboot**
`makefile.mon`	To build **osboot/dbg_core**

The **osboot** directory also contains the linker command files, each of which possess a base name corresponding to the target system for which it is intended. The linker command files' extensions have the following meanings:

- **.inc** files load object modules to build a relocatable **osboot**.
- **.mon** files load object modules to build relocatable **osboot** for linking with the MiniMON29K debugger core, **dbg_core**.
- **.lnk** files are used to produce an absolute image of **osboot** or **osboot/dbg_core**.

The sources of **osboot** are contained in the two subdirectories, **boot** and **traps**, under the **osboot** directory (see Figure 7-1 on page 7-3).

The **minimon** directory, which is at the same level as the **osboot** directory, contains the sources of the MiniMON29K debugger core, **dbg_core**. The **target** subdirectory under the **minimon** directory contains the assembler and C program source files.

This chapter describes how to build **osboot** in a number of different configurations. Table 7-2 lists the various configurations described in each section, along with the page number where their descriptions are found.

Table 7-2. Sample OSBOOT Configurations

This OSBOOT Configuration...	Is Described Here...
Building OSBOOT for Simulators	page 7-4
Building OSBOOT/DBG_CORE for Target Hardware Platforms	page 7-8
Building OSBOOT for Stand–Alone Systems	page 7-14

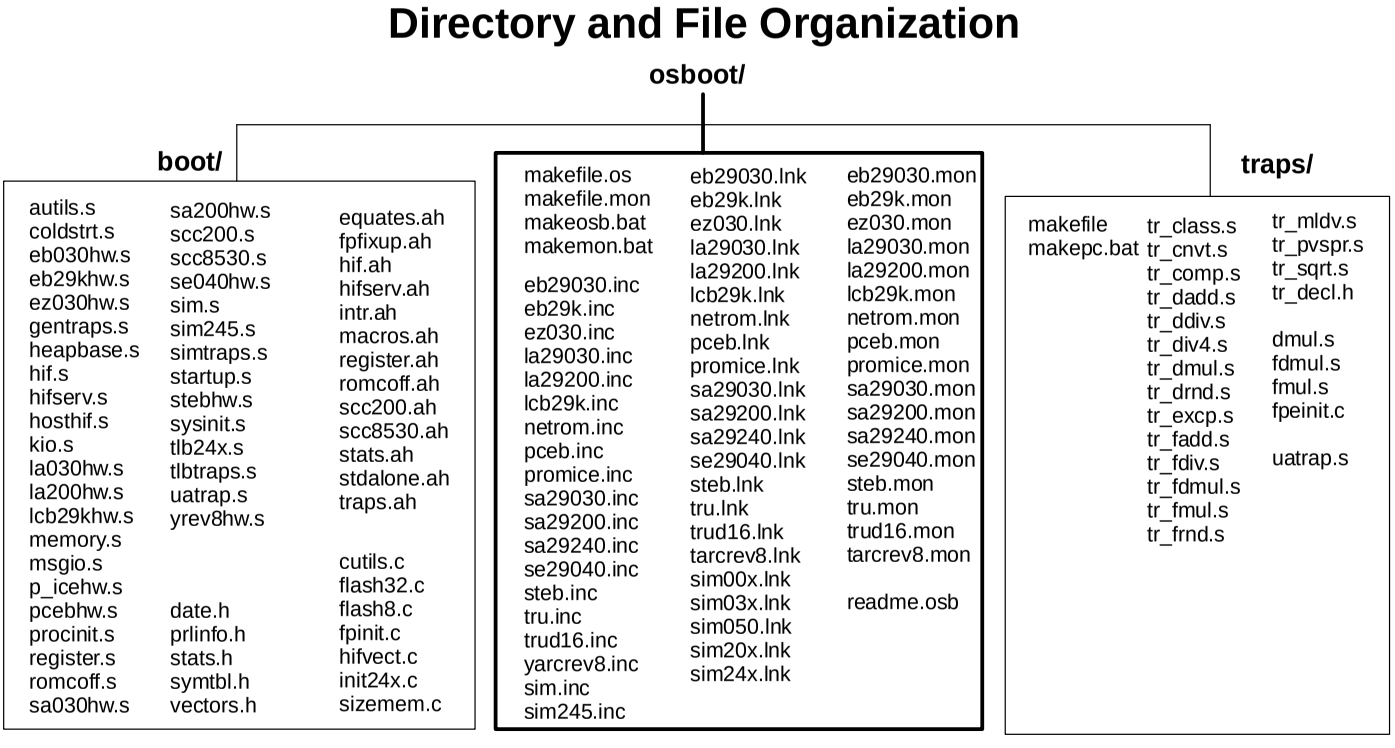

Figure 7-1. OSBOOT Directory and File Organization

Building OSBOOT for Simulators

Each member of the 29K Family of processors uses a specific version of **osboot** as its bootstrap code. AMD provides a set of utilities that lets you build different versions of **osboot** corresponding to different members of the 29K Family of processors. Depending on the value of a command-line parameter, versions of **osboot** can be built either separately or all at once. This section describes how to build **osboot** for simulators on MS-DOS and UNIX systems.

Location of OSBOOT Files Built for Simulators

Versions of **osboot** built for simulators are stored by the build program in a subdirectory named **sim** under the parent **osboot** directory. If the **sim** subdirectory does not already exist, the build program creates it. Figure 7-1 illustrates this directory structure. In this example, the **sim** subdirectory contains each possible version of **osboot** that can be built for simulators.

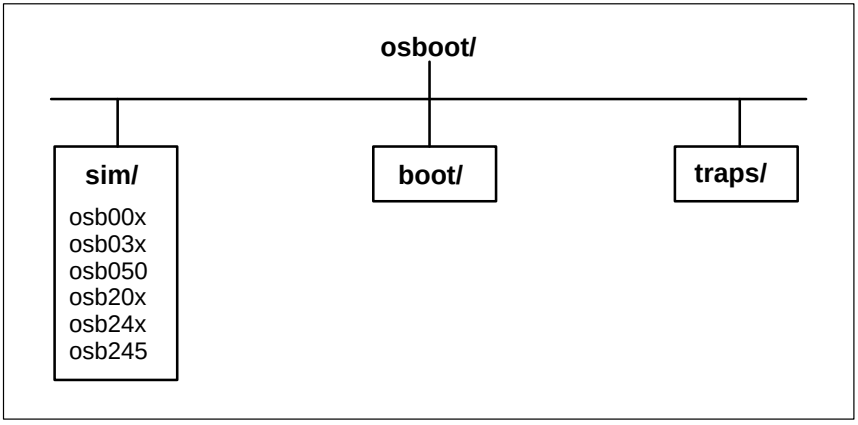

Figure 7-1. Subdirectory for Simulator Versions of OSBOOT

Linker Command Files Used to Build OSBOOT

A different linker command file is used to build **osboot** depending on the target processor type. Table 7-3 lists each linker command file with its corresponding **osboot** file and target processor type.

Table 7-3. Linker Command Files for Simulator Versions of OSBOOT

Linker Command File	OSBOOT File	Target Processor Type
sim00x.lnk	osb00x	Am29000, Am29005
sim03x.lnk	osb03x	Am29030, Am29035, Am29040
sim050.lnk	osb050	Am29050
sim20x.lnk	osb20x	Am29200, Am29205
sim24x.lnk	osb24x, osb25	Am29240, Am29243, Am29245

MS-DOS

The batch file to build **osboot** on an MS-DOS system is **makeosb.bat**. It is invoked in the **osboot** directory by specifying two arguments on the command line. The first argument must be the *board-name* and the second argument must be the *processor-name*. The linker command file corresponding to the *processor-name* specified is used for linking, as summarized in Table 7-3.

The *board-name* to use when building **osboot** for simulators is **sim**.

Syntax:

makeosb *board-name processor-name*

where:

board-name

Must be:

sim	For architectural and instruction set simulators

processor-name

Must be one of the following:

am2900x	For Am29000 and Am29005 processors
am2903x	For Am29030, Am29035, and Am29040 processors
am29050	For Am29050 processor
am2920x	For Am29200 and Am29205 microcontrollers
am2924x	For Am29240, Am29243, and Am29245 microcontrollers
all	To build a version of **osboot** for all members of the 29K Family

Example 1

To build **osboot** simulator files for all members of the 29K Family on an MS-DOS system, do the following:

1. Use the **cd** (change directory) command to make **osboot** the current directory.
2. Type the following command:
   ```
   makeosb sim all
   ```

A version of **osboot** for simulators is built for each different target processor and placed in the **/sim** directory (as in Figure 7-1 on page 7-4).

Example 2

Similarly, to build **osboot** simulator files for the Am29200 or Am29205 target processor (**osb20x**), do the following:

1. Use the **cd** (change directory) command to make **osboot** the current directory.
2. Type the following command:
   ```
   makeosb sim am2920x
   ```

UNIX

The make file to build **osboot** under UNIX systems is **makefile.os**. It is invoked in the **osboot/** directory by defining two variables on the command line: **proc** and **board**. The order of these definitions does not matter. The linker command files corresponding to the definition of the **proc** variable are used during linking.

The *board-name* to specify for simulators is **sim**.

Syntax: make –f *makefile.os* board=*board-name* proc=*processor-name*

where:

board-name

Must be:

sim	For architectural and instruction set simulators

processor-name

Must be one of the following:

am2900x	For Am29000 and Am29005 processors
am2903x	For Am29030 and Am29035 processors
am29040	For Am29040 processors
am29050	For Am29050 processors
am2920x	For Am29200 and Am29205 microcontrollers
am2924x	For Am29240, Am29243, and Am29245 microcontrollers
all	To build a version of **osboot** for all members of the 29K Family

Example 1

To build **osboot** /simulator files for all members of the 29K Family on a UNIX system, do the following:

1. Use the **cd** (change directory) command to make **osboot** the current directory.
2. Type the following command:
   ```
   make -f makefile.os board=sim proc=all
   ```

A version of **osboot** for simulators is built for each different target processor and placed in the **sim/** directory (as in Figure 7-1 on page 7-4).

Example 2

To build **osboot**/simulator files for the Am29240 target processors (**osb24x** and **osb245**), do the following:

1. Use the **cd** (change directory) command to make **osboot** the current directory.
2. Type the following command:
   ```
   make –f makefile.os board=sim proc=am2924x
   ```

Building OSBOOT/DBG_CORE for Target Hardware Platforms

AMD provides a set of utilities that lets you build different versions of **osboot**/**dbg_core** for use as the control program for different members of the 29K Family of processors. This section describes the procedures to build **osboot**/**dbg_core** on MS-DOS and UNIX systems.

Building OSBOOT/DBG_CORE—The General Procedure

The general build procedure for **osboot**/**dbg_core** is the same for all platforms. When creating **osboot**/**dbg_core**, you specify two command-line arguments. The first argument specifies the target for which this version of **osboot**/**dbg_core** will be created. The second argument specifies the 29K Family processor for which this version of **osboot**/**dbg_core** will be created.

The board name defined on the command line indicates the linker command file to be invoked at link time. Linker command files have the format *board_name*.**lnk**. The build program uses the board-name specified on the command line to establish naming conventions for the created files (and the directories in which they are stored). When the build command is successfully executed, the resulting **osboot**/**dbg_core** file is named *board_name*.**os**, and is stored in a directory named *board_name*. For example, specifying the board-name **sa29200** on the command line causes the linker command file **sa29200.lnk** to be invoked at link time. In turn, the resulting **osboot**/**dbg_core** file, **sa29200.os**, is stored in the **sa29200** directory. Figure 7-2 illustrates these naming conventions. Note that the illustration uses UNIX notation to indicate directories (forward slashes). The directory structure for an MS-DOS system is the same, but would be indicated with leading backward slashes (\).

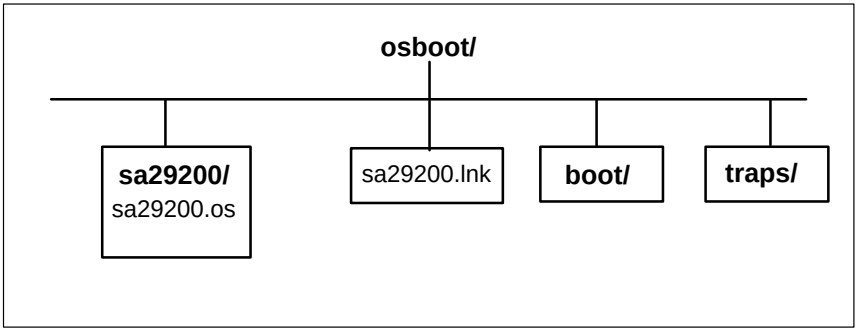

Figure 7-2. Naming Conventions for OSBOOT/DBG_CORE Files

Using OSBOOT/DBG_CORE

Once you have built **osboot/dbg_core**, you can use it as follows:

- For plug-in targets, such as the EB29K PC plug-in board or the YARC Rev 8 PC plug-in board, **osboot/dbg_core** files are downloaded and run on the writable ROM space of the target.
- For stand-alone targets (such as the EZ030 or SE29040 stand-alone boards), **osboot/dbg_core** files can be converted to a hexadecimal file that is then programmed to ROM. For example, the command below uses the **coff2hex** program to convert the text section of the **osboot/dbg_core** file **sa29200.os** to the Motorola S-records file **sa29200.hex**.

  ```
  coff2hex –t –c t sa29200.os
  ```

  The hexadecimal output file **sa29200.hex** can be used with a PROM programmer to create a MiniMON29K target PROM.

MS-DOS

The batch file used to build **osboot/dbg_core** on an MS-DOS system is **makemon.bat**. It is invoked in the **osboot** directory by specifying two arguments on the invocation line. The first argument must be the *board-name* and the second argument must be the *processor-name*. The linker command file corresponding to the *board-name* specified is used for linking.

The third argument, when specified, must follow the correct order. The third argument can be used to specify the baud rate to use for serial communications.

Syntax: makemon *board-name processor-name* [9600 | 38400]

where:

board-name

Must be one of the following:

eb29k	For AMD's EB29K PC plug-in board
yarcrev8	For the YARC Rev 8 PC plug-in board
lcb29k	For the YARC ATM (Sprinter) card
eb29030	For AMD's EB29030 PC plug-in board
ez030	For AMD's EZ030 stand-alone board
sa29200	For AMD's SA29200 and SA29205 stand-alone boards
steb	For STEP's STEB29K board
sa29240	For AMD's SE29240 stand-alone board
se29040	For AMD's SE29040 stand-alone board

processor-name

Must be one of the following:

am2900x	For the Am29000 and Am29005 processors
am2903x	For the Am29030 and Am29035 processors
am29040	For the Am29040 microprocessor
am29050	For the Am29050 processor
am2920x	For the Am29200 and Am29205 microcontrollers
am2924x	For Am29240, Am29243, and Am29245 microcontrollers

9600 | 38400

Must be the third argument, if specified. It specifies the baud rate to be used for serial communications. The default is 9600.

Example 1

To build **osboot/dbg_core** on a PC host for the SA29200 stand-alone target board, change directories to the **osboot** directory and enter:

```
makemon sa29200 am2920x 38400
```

This builds a COFF image of **osboot** and **dbg_core** for the SA29200 board and stores it in a file called **sa29200.os** in the **\sa29200** directory. The subdirectory **\sa29200** is created if it does not exist. The basename used for the final COFF image file corresponds to the first argument given, which is the target board's name, **sa29200**.

The third argument, 38400, specifies the baud rate to use for serial communications. The baud rate specified here is 38400, which overrides the default baud rate of 9600.

The COFF image, **sa29200\sa29200.os**, can be used to program EPROMs for the **sa29200** board.

Example 2

To build **osboot/dbg_core** on a PC host for the AMD EB29030 plug-in board, change directories to the **osboot** directory and enter:

```
makemon eb29030 am2903x
```

This builds a COFF image of **osboot** and **dbg_core** for the EB29030 board and stores it in a file called **eb29030.os** in the **\eb29030** directory. The subdirectory **\eb29030** is created if it does not exist. The name used for the final COFF image file corresponds to the target board's name, EB29030.

Because the EB29030 board is a plug-in target, no baud rate needs to be specified. Before debugging begins, the output file **eb29030.os** is downloaded and run on the target.

UNIX

The make file used to build the **osboot/dbg_core** is **makefile.mon.** It is invoked by defining two variables on the command line: **proc** and **board**. The order of these definitions does not matter. The linker command files corresponding to the definition of the **board** variable are used during linking.

Syntax: make -f *makefile.mon* board=*board-name*
proc=*processor-name*
[baudrate=38400 | baudrate=9600]

where:

board-name

Must be one of the following:

eb29k	For AMD's EB29K PC plug-in board
yarcrev8	For the YARC Rev 8 PC plug-in board
lcb29k	For the YARC ATM (Sprinter) card
eb29030	For AMD's EB29030 PC plug-in board
ez030	For AMD's EZ030 stand-alone board
sa29200	For AMD's SA29200 and SA29205 stand-alone boards
steb	For STEP's STEB29K board
sa29240	For AMD's SE29240 stand-alone board
se29040	For AMD's SE29040 stand-alone board

processor-name

Must be one of the following:

am2900x	For Am29000 and Am29005 processor
am2903x	For Am29030 and Am29035 processor
am29040	For Am29040 processor
am29050	For Am29050 processor
am2920x	For Am29200 and Am29205 microcontrollers
am2924x	For Am29240, Am29243, and Am29245 microcontrollers

baudrate=38400 | baudrate = 9600

Can be used to change the default baud rate to be used for serial communications. The default value is 9600.

Example 1

To build **osboot/dbg_core** on a UNIX host for an SA29200 stand-alone target board, change directories to the **osboot** directory, and enter:

```
make -f makefile.mon proc=am2920x board=sa29200 baudrate=9600
```

This builds a COFF image of **osboot/dbg_core**, and stores it in the **sa29200.os** file. It resides in an **sa29200/** directory, which is created if it does not exist. Notice that the final image's filename corresponds to the board specified on the command line. The baud rate is explicitly specified as 9600.

Next, you can use the **coff2hex** utility to convert selected sections (usually text sections) of the COFF file to a hex file format used to program EPROMs:

```
coff2hex -m -c t sa29200.os
```

The **sa29200.hex** file contains the Motorola S-records for the text sections of **sa29200.os** image to program EPROMs.

Program EPROMs on the SA29200 board with **sa29200.hex** and turn the power on.

Example 2

To build **osboot/dbg_core** on a UNIX host for the AMD EB29030 plug-in board, change directories to the **osboot** directory and enter:

```
make -f makefile.mon board=eb29030 proc=am2903x
```

This builds a COFF image of **osboot** and **dbg_core** for the EB29030 board and stores it in a file called **eb29030.os** in the **eb29030**/ directory. The subdirectory **eb29030/** is created if it does not exist. The name of the final COFF image file corresponds to the target board's name, EB29030.

Because the EB29030 board is a plug-in target, no baud rate needs to be specified. Before debugging begins, the **eb29030.os** output file is downloaded and run on the target.

Building OSBOOT for Stand-Alone Systems

This section provides instructions for building **osboot** for stand-alone systems. This includes both the stand-alone **osboot/application** model and the stand-alone **osboot/dbg_core/application** model.

Building **osboot** for stand-alone systems consists of linking a relocatable version of either **osboot** or **osboot/dbg_core** to a relocatable application program. Thus, before you can build the stand-alone system, you first need to build a relocatable version of the application program and either **osboot** or **osboot/dbg_core**.

This section is organized as follows:

- Building a Relocatable Version of OSBOOT
- Building a Relocatable Version of OSBOOT/DBG_CORE
- Building a Stand-Alone System – This section describes how to build a stand-alone system using the relocatable version of **osboot** or **osboot/dbg_core** that was built in the first two sections. The procedure is the same for either.

The "Examples" appendix provides detailed examples of how to port **osboot** to some sample stand-alone systems.

Building a Relocatable Version of OSBOOT

Building a relocatable version of **osboot** is almost exactly the same as building a version of **osboot** for simulators. You use the same DOS batch file (**makeosb.bat**) or the same UNIX make file (**makefile.os**), and use the same command-line arguments to specify both a *processor-name* and a *board-name* for the version of **osboot** to be built. The concepts of linker command files and file naming conventions are exactly the same as discussed in earlier sections.

The difference between building a relocatable version of **osboot** and any other version of **osboot** is that a **–DSTANDALONE** argument must be specified at the end of the command line. The **–DSTANDALONE** argument tells the build program to create an incrementally linked **osboot** module, *board-name*.o. This module can be linked with an application to produce a stand-alone system.

Build Syntax for MS-DOS and UNIX

The batch file used to build **osboot** on an MS-DOS system is **makeosb.bat**. The make file used to build osboot on a UNIX system is **makefile.os**. Both are invoked in the **osboot** directory by specifying two arguments on the invocation line.

DOS Syntax:

makeosb *board-name processor-name* –DSTANDALONE

For MS-DOS, the first argument must be the *board-name* and the second argument must be the *processor-name*.

UNIX Syntax:

make –f makefile.os board=*board-name* proc=*processor-name* standalone=–DSTANDALONE

where:

board-name

Must be one of the following:

ez030	For AMD's EZ030 stand-alone board
sa29200	For AMD's SA29200 and SA29205 stand-alone boards
steb	For STEP's STEB29K board
sa29240	For AMD's SE29240 stand-alone board
se29040	For AMD's SE29040 stand-alone board

processor-name

Must be one of the following:

am2900x	For the Am29000 and Am29005 processors
am2903x	For the Am29030 and Am29035 processors
am29040	For the Am29040 microprocessor
am29050	For the Am29050 processor
am2920x	For the Am29200 and Am29205 microcontrollers
am2924x	For Am29240, Am29243, and Am29245 microcontrollers

Example 1

To build a relocatable version of **osboot** on a PC host for the SA29200 stand-alone target board, change directories to the **osboot** directory and enter:

```
makeosb sa29200 am2920x –DSTANDALONE
```

The **makeosb** program builds the relocatable **osboot** file **sa29200.o** and stores it in the **\sa29200** directory.

Example 2

To build a relocatable version of **osboot** on a UNIX host for the AMD EZ030 board, change directories to the **osboot** directory and enter:

```
make –f makefile.os board=ez030 proc=am2903x standalone=–DSTANDALONE
```

The **makefile** program builds the relocatable **osboot** file **ez030.o** and stores it in the /**ez030** directory.

Building a Relocatable Version of OSBOOT/DBG_CORE

Building a relocatable version of **osboot/dbg_core** is almost exactly the same as building a version of **osboot/dbg_core** for a target hardware platform. You use the same DOS batch file (**makemon.bat**) or the same UNIX make file (**makefile.mon**), and use the same command-line arguments to specify both a *processor name* and a *board name* for the version of **osboot/dbg_core** to be built. The concepts of linker command files and file naming conventions are exactly the same as discussed in earlier sections.

The difference between building a relocatable version of **osboot/dbg_core** and any other version of **osboot/dbg_core** is that a **–DSTANDALONE** argument must be specified at the end of the command line. The **–DSTANDALONE** argument tells the build program to create an incrementally linked **osboot/dbg_core** module, **osbdbg.o**. This module can be linked with an application to produce a stand-alone system.

Build Syntax for MS-DOS and UNIX

The batch file used to build **osboot/dbg_core** on an MS-DOS system is **makemon.bat**. The make file used to build osboot on a UNIX system is **makefile.mon**. Both are invoked in the **osboot** directory by specifying the arguments summarized below on the command-line.

DOS Syntax:

makemon *board-name processor-name* [9600 | 38400] –DSTANDALONE

For MS-DOS, the order of the arguments must not change. The first argument must be the *board-name*, the second argument must be the *processor-name*, the baud rate must be the third argument, and **–DSTANDALONE** must be the last argument.

UNIX Syntax:

make –f makefile.os board=*board-name* proc=*processor-name* [baudrate=38400 | baudrate=9600] standalone=–DSTANDALONE

On a UNIX system, the order of the arguments is not relevant. The variables **proc**, **board**, and **standalone** must be defined. If the value of **baudrate** is not defined, the default value of 9600 is used.

NOTE: The build command for both MS-DOS and UNIX may cause the following error message, which can safely be ignored:

```
ERROR: (368) Unresolved external:
start
```

where:

board-name

Must be one of the following:

ez030	For AMD's EZ030 stand-alone board
sa29200	For AMD's SA29200 and SA29205 stand-alone boards
steb	For STEP's STEB29K board
sa29240	For AMD's SE29240 stand-alone board
se29040	For AMD's SE29040 stand-alone board

processor-name

Must be one of the following:

am2900x	For the Am29000 and Am29005 processors
am2903x	For the Am29030 and Am29035 processors
am29040	For the Am29040 microprocessor
am29050	For the Am29050 processor
am2920x	For the Am29200 and Am29205 microcontrollers
am2924x	For Am29240, Am29243, and Am29245 microcontrollers

9600 | 38400

Specifies the baud rate for serial communications.

Example 1

To build a relocatable version of **osboot/dbg_core** on a PC host for the SA29200 stand-alone target board, change directories to the **osboot** directory and enter:

```
makemon sa29200 am2920x 9600 –DSTANDALONE
```

The **makeosb** program builds the relocatable **osboot/dbg_core** file **osdbg.o** and stores it in the **\sa29200** directory. The baud rate for serial communications is set to 9600.

Example 2

To build a relocatable version of **osboot/dbg_core** on a UNIX host for the AMD EZ030 board, change directories to the **osboot** directory and enter:

```
make –f makefile.mon board=ez030 proc=am2903x
baudrate=38400 standalone=–DSTANDALONE
```

The **makefile** program builds the relocatable **osboot/dbg_core** file **osdbg.o** and stores it in the **/ez030** directory. The baud rate for serial communications is set to 38400.

Building a Stand-Alone System

The UNIX make files and MS-DOS batch files provided can be used to build a relocatable object module of **osboot**, which can be linked with your application program modules to produce a stand-alone system ready for programming EPROMs. The source files contain conditional code which is included when the manifest **STANDALONE** is defined on the command-line when compiling or assembling. The code is conditionally included for convenience. The code assumes no external loader, and therefore performs the following functions:

1. Zeroes out the BSS section and initializes data regions.
2. Assumes the **start** symbol to be defined as the entry point of the application program.
3. Executes the program in user mode without any translation.
4. Uses the assembler macro **$sizeof** to determine the size of the data region and sets up the heap base beyond that.

NOTE: Holes in the executable image could result in incorrect computation of the end address of the data region.

Most of the code specific to stand-alone system development is confined to the **stdalone.ah** header file, which is included in **hif.s**. In other places, code is conditionally included using the **.ifdef** STANDALONE construction.

The procedure to develop stand-alone systems involves the following steps:

1. Using the instructions in the previous two sections, build a relocatable version of either **osboot** or **osboot/dbg_core**. This procedure uses the example of a relocatable **osboot** module built for the EZ030 target. Accordingly, the name of the relocatable **osboot** file is **ez030.o**. In this example, the working directory is **\osboot**.
2. Compile your application program and make a relocatable object. For this example, the object is named **appl.o**.
3. Edit the linker command file provided for your target to suit your particular target application. Linker command files are named in the format *board-name*.**lnk**. So, for this example, the linker command file is **ez030.lnk**. You may need to add different data sections to the ORDER statement ordering the data sections. Otherwise, it should not require major editing for already known targets.

   NOTE: If you want to use the **LOAD** command to load object modules in the linker command file, you must specify those objects before the public definitions of the symbols contained in the default linker command file provided by AMD.

4. **First Link:** Using the edited linker command file for your target (in this example, the **ez030.lnk** linker command file), link the **osboot** module and your application program module using the compiler driver as shown below:

   ```
   hc29 -cmdez030.lnk ez030/ez030.o appl.o -o appl.out
   ```

   where **appl.out** contains the output.

5. Run the **romcoff** utility to produce the **RAMInit** routine to initialize data sections. In this example, for an EZ030 target, enter:

   ```
   romcoff -t -r appl.out raminit.o
   ```

   where **raminit.o** is the relocatable output object file containing the initialization routine. The **–r** option specifies that ROM space is readable; the **–t** option specifies to ignore text sections of the **appl.out** file. **raminit.o** contains routines that initialize DRAM and copy selected information from ROM to DRAM at run time. In most cases, you will elect to copy data sections (whose contents may be written during program execution) from ROM to DRAM before the program executes.

6. **Second Link:** Using the same edited linker command file generated for step 4 (above), relink the **osboot** and application modules with the newly created **raminit.o** module. In this example, for an EZ030 target, enter:

   ```
   hc29 -cmdez030.lnk ez030.o appl.o raminit.o -o appl.rom
   ```

   where **appl.rom** is the file containing the image. This linkage produces the final COFF file.

   NOTE: If you are loading the object modules in the linker command file, you may need to add the **raminit.o** object before creating the above link.

7. Use the **coff2hex** utility to convert selected sections (usually text sections) of the COFF file to a hex file format used to program EPROMs:

   ```
   coff2hex -m -c t appl.rom
   ```

   The **appl.hex** file contains the Motorola S-records for the text sections of **appl.rom** image to program EPROMs.

8. Program EPROMs with **appl.hex** and turn the power on.

OSBOOT Configuration

This section describes the configurable parameters of **osboot.** Configurable parameters can be defined differently according to the target system. The parameters are all defined as link-time constants and are defined in the linker command file. Functions of these parameters include:

- Defining the Current Processor Status Register (CPS) for the cold start process and for user programs.
- Defining a target system's memory configuration.
- Enabling and disabling dynamic memory sizing.
- Defining the execution location of **osboot** trap handlers.
- Defining the 29K Family processor's clock frequency.
- Defining ROM wait states, page mode DRAM, and SRAM memory.
- Defining serial port characteristics.

Table 7-4 lists the linker command filenames associated with each supported target system.

Table 7-4. Linker Command Filenames for Supported Targets

Target System	Linker Command Filename
EB29030	eb29030.lnk
EB29K	eb29k.lnk
EZ030	ez030.lnk
Laser29K-030	la29030.lnk
Laser29K-200	la29200.lnk
YARC ATM Sprinter	lcb29k.lnk
PCEB29K	pceb.lnk
SA29030	sa29030.lnk
SE29040	se29040.lnk
SA29200	sa29200.lnk
SE29240	sa29240.lnk
SA29205	sa29200.lnk
STEP's STEB	steb.lnk
YARC Rev 8	yarcrev8.lnk
Simulators	sim00x.lnk, sim03x.lnk, sim050.lnk, sim20x.lnk, sim24x.lnk

NOTE: Some of the boards in Table 7-4 are no longer available commercially, but are still in use. Note also that the linker command filenames for some targets do not correspond directly to the target's name.

The source files refer to these link-time parameters as external variables. To pass the values defined in the command file to the source files, the linker command, **PUBLIC**, must be used to define values for each parameter. The syntax of the **PUBLIC** linker command is:

PUBLIC *symbol = value*

where *symbol* is a user-defined external definition symbol, and *value* is the value of the symbol. The symbol names specified by the linker's **PUBLIC** command take precedence over symbol names defined during assembly.

The linker command files for the target hardware systems supported by AMD are provided. They define the default values for those target systems.

The list of configuration parameters that must be defined for all targets in the linker command files is shown in Table 7-5. Some of these parameters are not used in some target system configurations. However, they still need to be defined to avoid getting an unresolved external link time error. The values of symbols that are ignored or which do not apply to the target system can be zero.

Table 7-5. Configuration Parameters

Configuration Parameter	Meaning
These parameters apply to both 2-bus and 3-bus microprocessors.	
init_CPS	The value used to initialize the CPS register at the beginning of the cold start process.
DMemStart	The starting address of the data memory space. This is used to initialize the VAB register. It is also the base address used when dynamically sizing the external data memory at run-time. It can be the same as **IMemStart** in cases where there is a single linear memory space.
DMemSize	The maximum possible size of the data memory region.
TicksPerMillisecond	This must be defined to the processor clock frequency in ticks per millisecond. It is used by the HIF kernel services of **osboot**.
ClockFrequency	This must be defined to the processor clock frequency. It is used by the HIF kernel services of **osboot**.
These parameters only apply to 3-bus microprocessors.	
IMemStart	The starting address of the instruction memory space.
IMemSize	The maximum possible size of the instruction memory region.
RMemStart	The starting address of the ROM address space.
RMemSize	The maximum possible size of the ROM region.

Configuration Parameter	Meaning
TRAPINROM	Set to 0x2 (two) to specify that the trap handlers reside in the ROM address region, or 0x0 (zero) to specify that the trap handlers reside in the instruction memory region. Trap handlers should only be located in ROM space (that is, **TRAPINROM** set to 0x2) for those 29K Family microprocessors using 3-bus architecture. For other members of the 29K Family, trap handlers should be located in instruction memory space (**TRAPINROM** set to 0x0).
This parameter only applies to 2-bus microprocessors.	
EnableDRAMSizing	If set to 1, DRAM memory range is determined at run time. The memory range found is used to configure the DRAM Controller. If set to 0, the value specified by the **DMemSize** variable is used.

Table 7-6 lists the **osboot** configuration parameters that must be defined if the target contains an 85C30 (Serial Communications Controller) device. If these parameters are defined in the command file of a target system that does not require them, they will be ignored.

Table 7-6. Configuration Parameters

Configuration Parameter	Meaning
SCC8530_CHA_CONTROL	This value is used as the address to access the control register of Channel A of the 85C30 device on the target system.
SCC8530_CHB_CONTROL	This value is used as the address to access the control register of Channel B of the 85C30 device on the target system.
SCC8530_CHA_DATA	This value is used as the address to access the data register of Channel A of the 85C30 device on the target system.

Configuration Parameter	Meaning
SCC8530_CHB_DATA	This value is used as the address to access the data register of Channel B of the 85C30 device on the target system.
SCC8530_BAUD_CLK_ENBL	This value specifies the enabling/disabling of the baud rate clock generator of the 85C30 device on the target system.

Table 7-7 lists those parameters that must be defined for a micro*controller* target. These parameters initialize ROM and DRAM controllers. Because micro*processor* targets do not have ROM or DRAM controllers, these parameters do not apply to them. However, to avoid unresolved external link time errors, you should be sure to set them even if your target is a 29K Family microprocessor. The values of symbols that do not apply to the target system are ignored. They can safely be set to zero.

Table 7-7. Configuration Parameters

Configuration Parameter	Meaning
RMCT_VALUE	The value used to initialize the ROM Control register of the 29K Family microcontrollers.
DRCT_VALUE	The value used to initialize the DRAM Control register of the 29K Family microcontrollers.
RMCF_VALUE	The value used to initialize the ROM Configuration register of the 29K Family microcontrollers.
UCLK	Specifies the clock frequency controlling the UART. Used to compute the Baud Rate Divisor.

Sample Linker Command File

To demonstrate the implementation of the parameters described in the previous tables, the following code sample is provided. It shows the contents of the **sa29200.lnk** linker command file provided with the **osboot** software. The comment portions of this file have been removed to simplify reading. To see the comment portions of this file, use a text editor to view the **sa29200.lnk** file in the **osboot** directory.

```
ALIGN  ProcInit=16
ORDER  Reset=0x0
ORDER  ProcInit,Osbtext,.text,!text
ORDER  lit,!lit
ORDER  vectable=0x40000000
ORDER  msg_data=0x40000400
ORDER  .data,!data
ORDER  OsbBss,dbg_030,dbg_bss,cfg_bss,.bss,!bss
ORDER  HeapBase
ORDER  .comment

; Defines initial value of CPS register
public  _init_CPS=0x87F

; Defines target system's memory configuration
public  VectorBaseAddress=0x40000000
public  IMemStart=0x40000000
public  IMemSize=0xffffff
public  DMemStart=0x40000000
public  DMemSize=0xffffff
public  RMemStart=0x0
public  RMemSize=0xffffff

; Enables/Disables dynamic memory sizing
public  EnableDRAMSizing=1

; Defines ROM wait states,page mode DRAM,SRAM memory
public  RMCT_VALUE=0x03030303
public  DRCT_VALUE=0x888800FF
public  RMCF_VALUE=0x00f8f8f8

; Defines execution location of trap handlers
public  _TRAPINROM=0

; Defines 29K Family processor clock frequency
public  TicksPerMillisecond=16000
public  ClockFrequency=16000000

; Defines serial port characteristics.
public UCLK=32000000
public INITBAUD=9600

public  SCC8530_CHA_CONTROL=0xC0000007
public  SCC8530_CHB_CONTROL=0xC0000003
public  SCC8530_CHA_DATA=0xC000000F
public  SCC8530_CHB_DATA=0xC000000B
public  SCC8530_BAUD_CLK_ENBL=3
```

Appendix A

Examples

This appendix provides examples of some common **osboot** configuration models. Most of the examples in this appendix follow procedures detailed elsewhere in this manual. The appendix provides the following examples:

- Building the Stand-Alone OSBOOT/Application Model for AMD's SE29240 board on page A-2.
- Building the Stand-Alone OSBOOT/DBG_CORE/Application Model for AMD's SE29040 board on page A-4.
- Building the OSBOOT/Application Model to transfer from ROM to SRAM for testing on page A-6.
- Building the OSBOOT/Application Model to transfer from ROM to DRAM for testing on page A-9.
- Building the OSBOOT/DBG_CORE Model for a system without DRAM on page A-11.

Building the Stand-Alone OSBOOT/Application Model for the SE29240

This example shows how to build the Stand-Alone OSBOOT/Application Model for AMD's SE29240 stand-alone board. This example follows the same procedure detailed in "Building OSBOOT for Stand-Alone Systems," starting on page 7-14.

In this example, the baud rate used for serial communications is 9600 baud.

1. Use the **cd** command to change to the **...\29k\osboot** directory.
2. Build a relocatable version of **osboot** for the SE29240 board. Use the following command lines for MS-DOS and UNIX, respectively:

   For MS-DOS:
   ```
   makeosb sa29240 am2924x 9600 –DSTANDALONE
   ```

   For UNIX:
   ```
   make –f makefile.os board=sa29240 proc=am2924x
   baudrate=9600 standalone=–DSTANDALONE
   ```

   Regardless of the operating system, the build program creates the relocatable **osboot** file, **sa29240.o**, and stores it in the directory **sa29240**.

3. In this example, the name of the application program is **appl.c**. Use the **hc29** program to build a relocatable version of **appl.c** using the following command:

   ```
   hc29 –c –o appl.o appl.c
   ```

   The **hc29** program creates the relocatable application file **appl.o**.

4. Edit the linker command file. Because AMD provides a linker command file to work with the SE29240 board (**sa29240.lnk**), there is no need to edit the linker command file in this case. When building **osboot** for targets for which AMD does not supply a custom linker command file, you will need to edit a linker command file to work with your target.
5. **First link.** Use the **hc29** program to link the relocatable **osboot** module built in Step 2 with the relocatable application module built in Step 3 using the following command.

   ```
   hc29 –cmdsa29240.lnk sa29240/sa29240.o appl.o –o appl.out
   ```

The **hc29** program creates the COFF image file **appl.out**. The **appl.out** file represents the linkage of the relocatable **osboot** file, **sa29240.o**, and the relocatable application file, **appl.o**.

6. Use the **romcoff** utility to create the **raminit.o** module for **appl.out**. The **raminit.o** module is used to initialize DRAM and copy selected sections of the **appl.out** file from ROM to DRAM before program execution starts. In this example, **raminit.o** copies images of the **.data** and **.bss** sections of the **appl.out** file stored in ROM to DRAM at run time. Because the **–t** argument is specified in the command line below, text sections are excluded from **raminit.o**. The **–r** option specifies that ROM space is readable. Use the following command:

   ```
   romcoff –t –l –r appl.out raminit.o
   ```

7. **Second link.** Use the **hc29** program to link **raminit.o** with the relocatable version of **osboot** and the relocatable version of the application program. Use the same linker command as in Step 5, as follows:

   ```
   hc29 –cmdsa29240.lnk sa29240/sa29240.o appl.o raminit.o
   –o appl.rom
   ```

   The **hc29** program creates the **appl.rom** COFF file.

8. Use the **coff2hex** command to convert the text section of the COFF file, **appl.rom**, to a hex file that can be used to program EPROMs on the target. Use the following command syntax:

   ```
   coff2hex –m –c tl appl.rom
   ```

   The **coff2hex** command creates the hex file **appl.hex**. Use this file to program EPROMs. Then, install the EPROMs on the target and turn on the power.

Building the Stand-Alone OSBOOT/DBG_CORE/ Application Model for the SE29040

This example shows how to build the Stand-Alone OSBOOT/DBG_CORE/Application Model for AMD's SE29040 stand-alone board. This example follows the same procedure described in, "Building OSBOOT for Stand-Alone Systems," starting on page 7-14.

In this example, the baud rate used for serial communications is 38400 baud.

1. Use the **cd** command to change to the **.../29k/osboot** directory.
2. Build a relocatable version of **osboot/dbg_core** for the SE29040 board. Use the following command lines for MS-DOS and UNIX, respectively:

   For MS-DOS:
   ```
   makemon se29040 am29040 38400 –DSTANDALONE
   ```

   For UNIX:
   ```
   make –f makefile.mon board=se29040 proc=am29040
   baudrate=38400 standalone=–DSTANDALONE
   ```

   Regardless of the operating system, the build program creates the relocatable **osboot/dbg_core** file, **osbdbg.o**, and stores it in the **se29040** directory.

   NOTE: The build command for both MS-DOS and UNIX may cause the following error message, which can safely be ignored:

   ```
   ERROR: (368) Unresolved external:
   start
   ```

3. In this example, the name of the application program is **appl.c**. Use the **hc29** program to build a relocatable version of **appl.c** using the following command:

   ```
   hc29 –c –o appl.o appl.c
   ```

   The **hc29** program creates the relocatable application file **appl.o**.

4. Edit the linker command file. Because AMD provides a linker command file to work with the SE29040 board (**se29040.lnk**), there is no need to edit the linker command file in this case. When building **osboot** for targets for which AMD does not supply a custom linker command file, you will need to edit a linker command file to work with your target.

5. **First link.** Use the **hc29** program to link the relocatable **osboot/dbg_core** module built in Step 2 with the relocatable application module built in Step 3 using the following command:

   ```
   hc29 –cmdse29040.lnk se29040/osbdbg.o appl.o –o appl.out
   ```

   The **hc29** program creates the COFF image file **appl.out**. The **appl.out** file represents the linkage of the relocatable **osboot/dbg_core** file, **osbdbg.o**, and the relocatable application file, **appl.o**.

6. Use the **romcoff** utility to create the **raminit.o** module for **appl.out**. The **raminit.o** module is used to initialize DRAM and copy selected sections of the **appl.out** file from ROM to DRAM before program execution starts. In this example, **raminit.o** copies images of the **.data** and **.bss** sections of the **appl.out** file stored in ROM to DRAM at run time. Because the **–t** argument is specified in the command line below, text sections are excluded from **raminit.o**. The **–r** option specifies that ROM space is readable.
   Use the following command:

   ```
   romcoff –t –l –r appl.out raminit.o
   ```

7. **Second link.** Use the **hc29** program to link **raminit.o** with the relocatable version of **osboot/dbg_core** and the relocatable version of the application program. Use the same linker command as in Step 5, as follows:

   ```
   hc29 –cmdse29040.lnk se29040/osbdbg.o appl.o raminit.o –o appl.rom
   ```

   The **hc29** program creates the COFF file **appl.rom**.

8. Use the **coff2hex** command to convert the text section of the COFF file, **appl.rom**, to a hex file that can be used to program EPROMs on the target. Use the following command syntax:

   ```
   coff2hex –m –c tl appl.rom
   ```

   The **coff2hex** command creates the hex file **appl.hex**. Use this file to program EPROMs. Then, install the EPROMs on the target and turn on the power.

Building the OSBOOT/Application Model to Transfer from ROM to SRAM

This example shows how to build the OSBOOT/Application Model for AMD's SA29200 stand-alone board. In this example, however, instead of using the linker command file as supplied for the SA29200 board, the linker command file is edited so that the COFF file created by linking relocatable versions of **osboot** and an application file can be executed from SRAM instead of ROM. Then, the example uses the **mondfe** debugger front end to "yank" the COFF file to the SRAM of the target SA29200 board for testing purposes.

This example is very similar to the procedure described in "Building OSBOOT for Stand-Alone Systems," starting on page 7-14.

1. Use the **cd** command to change to the **.../29k/osboot** directory.
2. Create a working directory to contain edited versions of master template files provided by AMD. In this example, the **sa2920s8** directory is created.
3. To link **osboot** with an application, both **osboot** and the application need to be relocatable. Build a relocatable version of **osboot** for the SA29200 board. Use the following command lines for MS-DOS and UNIX, respectively:

   For MS-DOS:
   ```
   makeosb sa29200 am29200x –DSTANDALONE
   ```

   For UNIX:
   ```
   make –f makefile.os board=sa29200 proc=am29200x
   standalone=–DSTANDALONE
   ```

   Regardless of the operating system, the build program creates the relocatable **osboot** file, **sa29200.o**, and stores it in the **sa29200** directory.

4. Edit the linker command file provided by AMD for the SA29200 board so that the COFF application file created using this linker command file can be run in SRAM instead of ROM. The linker command file provided by AMD for the SA29200 board is called **sa29200.lnk** and is located in the **osboot** directory**.**

   a. Make a copy of the **sa29200.lnk** file in the working directory created in Step 1 (**sa2920s8**).

b. Using a text editor, open the **sa29200.lnk** file and change the entry point of the **osboot** module from “0x0” (ROM) to “0x80000” (SRAM). This is done by changing the line that reads, “ORDER Reset=0x0,” to “ORDER Reset=0x80000.” Save the file. In this example, the file is saved as **sa2920s8.lnk**. You can see the contents of the unedited **sa29200.lnk** file in “Sample Linker Command File,” on page 7-25.

c. **First link.** Use the **hc29** program to build a relocatable version of the application program to be linked with the relocatable version of **osboot** built in Step 3. In this example, the name of the application file is **appl.c**. The following **hc29** command builds a relocatable version of **appl.c** (called **appl.o**) and then uses the linker command file we created in Step 4b (**sa2920s8**) to link **appl.o** with the relocatable **osboot** module **sa29200.o**:

```
hc29 –cmd./sa2920s8.lnk ../sa29200/sa29200.o appl.c –o sa2920s8.out
```

The **hc29** program creates the COFF image file **sa2920s8.out**. The **sa2920s8.out** file is the linkage of the relocatable **osboot** file, **sa29200.o**, and the relocatable application file **appl.o**.

5. Use the **romcoff** utility to create the **raminit.o** module for **sa2920s8.out**. The **raminit.o** module is used to initialize DRAM and copy selected sections of the **sa2920s8.out** file from SRAM to DRAM before program execution starts. In this example, **raminit.o** copies images of the **.data** and **.bss** sections of the **sa2920s8.out** file stored in SRAM to DRAM at run time. Because the **–t** argument is specified in the command line below, text sections are excluded from **raminit.o**. The **–r** option specifies that ROM space is readable.

   Use the following command:

   ```
   romcoff –t –l –r sa2920s8.out raminit.o
   ```

6. **Second link.** Use the **hc29** program to link **raminit.o** with the relocatable version of **osboot** and the relocatable version of the application program. Use the same linker command as in Step 4

   ```
   hc29 –cmdsa2920s8.lnk sa29200/sa29200.o appl.o raminit.o –o sa2920s8.rom
   ```

   The **hc29** program creates the COFF file **sa2920s8.rom**.

7. Start **mondfe** in **Supervisor** mode and “yank” the file **sa2920s8.rom** to SRAM. Use the following command:

   ```
   mondfe –TIP serial96S sa2920s8.rom
   ```

For complete information on using **mondfe**, see the *MiniMON29K User Interface: MONDFE* manual.

At this point, you can begin executing and debugging the program using **mondfe**. When the program is fully debugged, the stand-alone OSBOOT/Application program can be built and programmed to EPROMs.

Building the OSBOOT/Application Model to Transfer from ROM to DRAM

This example shows how to build the Stand-Alone OSBOOT/Application Model for AMD's SA29200 stand-alone board. In this example, however, the **raminit.o** module is created so that the entire OSBOOT/Application module (including both the text and data sections) is transferred from ROM to DRAM at power-on (or when **RESET** is asserted) and executed there.

This example follows the same procedure described in, "Building OSBOOT for Stand-Alone Systems," starting on page 7-14.

In this example, the baud rate used for serial communications is 9600 baud.

1. Use the **cd** command to change to the **.../29k/osboot** directory.
2. Build a relocatable version of **osboot** for the SA29200 board. Use the following command lines for MS-DOS and UNIX, respectively:

   For MS-DOS:
   ```
   makeosb sa29200 am2920x 9600 –DSTANDALONE
   ```

   For UNIX:
   ```
   make –f makefile.os board=sa29200 proc=am2920x
   baudrate=9600 standalone=–DSTANDALONE
   ```

   Regardless of the operating system, the build program creates the relocatable **osboot** file, **sa29200.o**, and stores it in the **sa29200** directory.

3. In this example, the name of the application program is **appl.c**. Use the **hc29** program to build a relocatable version of **appl.c** using the following command:

   ```
   hc29 –c –o appl.o appl.c
   ```

   The **hc29** program creates the relocatable application file **appl.o**.

4. Edit the linker command file. Because AMD provides a linker command file to work with the SA29200 board (**sa29200.lnk**), there is no need to edit the linker command file in this case. When building **osboot** for targets for which AMD does not supply a custom linker command file, you will need to edit a linker command file to work with your target.

5. **First link.** Use the **hc29** program to link the relocatable **osboot** module built in Step 2 with the relocatable application module built in Step 3 using the following command:

   ```
   hc29 –cmdsa29200.lnk sa29200/sa29200.o appl.o –o appl.out
   ```

   The **hc29** program creates the COFF image file **appl.out**. The **appl.out** file is the linkage of the relocatable **osboot** file, **sa29200.o**, and the relocatable application file, **appl.o**.

6. Use the **romcoff** utility to create the **raminit.o** module for **appl.out**. The **raminit.o** module is used to initialize DRAM and copy selected sections of the **appl.out** file from ROM to DRAM before program execution starts. In previous examples, the **romcoff** utility was used with the **–t** argument so that text sections of the input file (in this case, **appl.out**) were not included in the output file (**raminit.o**). In this example, we want the entire application to execute from DRAM. Accordingly, we omit the **–t** argument from the command line so that the entire input file is included in the output file and will be executed from DRAM. The **–r** option specifies that ROM space is readable.

   Use the following command:

   ```
   romcoff –r appl.out raminit.o
   ```

7. **Second link.** Use the **hc29** program to link **raminit.o** with the relocatable version of **osboot** and the relocatable version of the application program. Use the following command:

   ```
   hc29 –nocrt0 –cmdsa29200.lnk sa29200/sa29200.o appl.o raminit.o –o
   appl.rom
   ```

   Notice that this command line specifies the compiler option “–nocrt0.” This is done to avoid compiler error messages that might occur because of the presence of the **crt0** in the **raminit.o** file. Because in this example **raminit.o** contains the text section of **appl.out**, it also contains the **crt0** from the first link in Step 5. The **hc29** program creates the COFF file **appl.rom**.

8. Use the **coff2hex** command to convert the COFF file, **appl.rom**, to a hex file that can be used to program EPROMs on the target. Use the following command syntax:

   ```
   coff2hex –m –c t appl.rom
   ```

The **coff2hex** command creates the hex file **appl.hex**. Use this file to program EPROMs. Then, install the EPROMs on the target and turn on the power.

Building OSBOOT/DBG_CORE for a System Without DRAM

This example shows how to build **osboot/dbg_core** for a system that does not have any DRAM. In this example, we use an SA29200 target that has only ROM and 512 Kbyte of byte-writable SRAM. The example demonstrates how to customize the linker command file so that a version of **osboot/dbg_core** for a system without DRAM (SRAM only) is built.

In this example, the baud rate used for serial communications is 9600 baud (the default).

1. Use the **cd** command to change to the **.../29k/osboot** directory.
2. When the build program creates **osboot/dbg_core** for the SA29200 target, it uses a linker command file that specifies the default values for configurable parameters of **osboot**. AMD provides a default linker command file for the SA29200 target called **sa29200.lnk**. For this example, we edit **sa29200.lnk** so that the configurable parameters of **osboot** are set to accommodate a target that has SRAM but no DRAM.

   The following code section shows the **sa29200.lnk** file, as edited to support an SRAM-only target. Each edited line is indicated by the comment field reading "**;** SRAM." A detailed explanation of each change (by line number) follows the code section.

```
1       ALIGN   ProcInit=16
2       ORDER   Reset=0x0
3       ORDER   ProcInit,Osbtext,.text,!text
4       ORDER   .lit,!lit
5       ORDER   vectable=0x80000         ; SRAM
6       ORDER   msg_data=0x80400         ; SRAM
7       ORDER   .data,!data
8       ORDER   OsbBss,dbg_030,dbg_bss,cfg_bss,.bss,!bss
9       ORDER   HeapBase
10      ORDER   .comment

11      ; Defines initial value of CPS register
12      public  _init_CPS=0x87F
```

```
13      ; Defines target system's memory configuration
14      public  VectorBaseAddress=0x8000 ; SRAM
15      public  IMemStart=0x40000000
16      public  IMemSize=0xffffff
17      public  DMemStart=0x80000               ; SRAM
18      public  DMemSize=0x7ffff        ; SRAM
19      public  RMemStart=0x0
20      public  RMemSize=0xffffff

11      ; Defines initial value of CPS register
12      public  _init_CPS=0x87F

13      ; Defines target system's memory configuration
14      public  VectorBaseAddress=0x0x80000     ; SRAM
15      public  IMemStart=0x40000000
16      public  IMemSize=0xffffff
17      public  DMemStart=0x80000               ; SRAM
18      public  DMemSize=0x7ffff        ; SRAM
19      public  RMemStart=0x0
20      public  RMemSize=0xffffff

21      ; Enables/Disables dynamic memory sizing
22      public  EnableDRAMSizing=0              ; SRAM

23      ; Defines ROM wait states, page mode DRAM, SRAM
          memory
24      public  RMCT_VALUE=0x0b030303           ; SRAM
25      public  DRCT_VALUE=0x88880000           ; SRAM
26      public  RMCF_VALUE=0x0008f8f8           ; SRAM

27      ; Defines execution location of trap handlers
28      public  _TRAPINROM=0

29      ; Defines 29K processor clock frequency
30      public  TicksPerMillisecond=16000
31      public  ClockFrequency=16000000

32      ; Defines serial port characteristics.
33      public UCLK=32000000
34      public INITBAUD=9600

35      public  SCC8530_CHA_CONTROL=0xC0000007
36      public  SCC8530_CHB_CONTROL=0xC0000003
37      public  SCC8530_CHA_DATA=0xC000000F
38      public  SCC8530_CHB_DATA=0xC000000B
39      public  SCC8530_BAUD_CLK_ENBL=3
```

Each change made to the **sa29200.lnk** file to support an SRAM-only target is described by line-number below.

a. In lines 5, 6, 14, and 17, memory addresses are changed from DRAM-base addresses to SRAM-base addresses.

b. The value of the **EnableDRAMSizing** variable in line 22 determines whether dynamic memory sizing is enabled. Because dynamic memory sizing is only supported by a DRAM system, **EnableDRAMSizing** is changed to **0** (disabled). Without dynamic memory sizing available, the system cannot determine available memory by itself. Therefore, the **DMemSize** variable in line 18 must be set to the exact memory size on the target (512 Kbyte; 0x7ffff in hex).

c. On line 24, the BWE bit (bit 27) of the RMCT register needs to be set to 1 to support byte-writable SRAM. In hex, the value of the **RMCT_VALUE** variable is changed from 0x03030303 to 0x0b030303.

d. Because there is no DRAM on the target, the **REFRATE** (refresh rate) field of the DRCT register is disabled and set to 0. In hex, the value of the **DRCT_VALUE** variable (line 25) is changed from 0x888800FF to 0x88880000.

e. On line 26, the value of the **RMCF_VALUE** variable is changed from 0x00f8f8f8 to 0x0008f8f8 to define the SRAM memory bank in the RMCF register. The value 0x0008f8f8 indicates that the starting address of the SRAM memory on the second bank is 0x80000 and the memory size is 512 Kbyte.

Detailed information on how to set the Am29200 microcontroller's registers can be found in the *Am29200 and Am29205 RISC Microcontroller User's Manual*.

3. At this point, you are ready to build **osboot/dbg_core** for the target using the linker command file we edited in the previous step. Use the following command lines for MS-DOS and UNIX, respectively:

   For MS-DOS:
   ```
   makemon sa29200 am29200x
   ```

   For UNIX:
   ```
   make –f makefile.mon board=sa29200 proc=am29200x
   ```

   Regardless of the operating system, the build program creates a COFF image of **osboot** and **dbg_core** for the SA29200 target using the customized settings in the **sa29200.lnk** linker command file and stores it in a file called **sa29200.os** in the **sa29200** directory.

4. Use the **coff2hex** utility to create the hex file **sa29200.hex** from the COFF file **sa29200.os**, as follows.

```
coff2hex –r –t sa29200/sa29200.os –o sa29200.hex
```

5. Program EPROMs with **sa29200.hex** and turn the power on.

Appendix B

Using the HIF IOCTL Service for Non-Blocking Reads

The Host Interface (HIF) kernel of Release 3.0 or later of the MiniMON29K product adds support for nonblocking read operations. This new feature allows application programs to continue processing while waiting for input data to be transferred. (To use this feature, MiniMON29K Release 3.0 or later must be loaded on both the target and the host.)

This appendix shows example code that demonstrates how a nonblocking read can be used to create an interactive menu that allows subsequent processing to continue while waiting for user input.

The Problem

The default input mode used by the HIF kernel of the MiniMON29K product is COOKED (0x0000), which blocks (suspends) the application program issuing a **read** operation from the standard input device (terminal) until the input data has been transferred.

While blocked mode may be effective for some applications, it makes it difficult to implement and debug interactive menu-driven applications that require processing to continue while waiting for user input.

The Solution

The HIF kernel of the MiniMON29K Release 3.0 (or later) product implements nonblocking read support for standard input devices (terminals). Using the HIF **ioctl** service, the input mode of the standard input device can be changed from COOKED to NBLOCK mode, which allows the application program to continue processing while waiting for input data to be transferred.

The HIF IOCTL Service

The HIF **ioctl** service establishes the operation mode of a specified file or device. It is intended to be applied primarily to terminal-like devices; however, certain modes apply to mass-storage files or to other related input/output devices.

In COOKED (0x0000) mode (the default input mode), when a **read** operation for a terminal-like device is issued by the application program, the kernel blocks (suspends) any further execution of the application program until the data has been transferred. Using the HIF **ioctl** service, the input mode of the standard input device can be set to NBLOCK (0x0010) mode, which specifies that subsequent read operations be executed without suspending (blocking) the application program issuing the **read** request.

The HIF READ Service

After setting standard input to NBLOCK mode, a HIF **read** operation on the standard input device returns immediately to the application program. The return value of the **read** service contains the number of characters currently available, or -1 if none are available. The application program examines the return value from the **read** service to determine if any input data is available. If the return value is -1, the application program continues other processing while waiting for the input data.

Example Code

Following is a short code example showing how the technique explained above can be used for the creation of an interactive menu. The code in boldface highlights the use of the HIF **ioctl** service.

```
#include <stdio.h>
#include <hif.h>
#include <stdlib.h>
showmen() {
/*************************************************************************/
/* The following subroutine will display a menu to standard output. The  */
/* _write() HIF service was used rather than printf for the following    */
/* reasons:                                                              */

/*   - To show the usage of the _write() HIF service                     */
/*   - When characters are sent to standard output using printf, the output */
/*     is buffered and will not be displayed until a '\n' is used.       */
/*     Therefore, the "Enter your selection ->" line would not appear on */
/*     standard output if printf were used because there is no '\n'      */
/*     character at the end of the string                                */
/*************************************************************************/
  _write(1,"\nMenu:\n",7);
  _write(1,"\t1) Selection #1\n",17);
  _write(1,"\t2) Selection #2\n",17);
  _write(1,"\t3) Exit\n",9);
  _write(1,"\nEnter your selection -> ",25);
}
void main() {
  /* Declare necessary variables */
  int terminate=0, proc=0;
  /* Create a 1-character buffer to hold input */
  char *input=(char *) malloc(sizeof(char));
  /* Set standard input to NBLOCK mode using the _ioctl() HIF service */
  _ioctl(0,0x0010);

  showmen();
  /*************************************************************************/
  /* Following is the main loop where processing occurs. The exit          */
  /* condition is set by selecting the appropriate menu item.              */
  /*************************************************************************/
  while(!terminate) {

    /*************************************************************************/
    /* Because standard input has been set to NBLOCK mode using the          */
    /* _ioctl() HIF service, the _read() HIF service will return a negative */
    /* number if there is no input available. If there is input available,  */
    /* _read() will return a non-negative number and the character read     */
    /* will be contained in the input buffer.                                */
    /*************************************************************************/
    if(_read(0,input,1)>=0) {
      /* React to the input received */
      switch(input[0]) {
        case '1':
          printf("\n\nYou selected choice #1.\n");
          showmen();
          break;
```

```
        case '2':
          printf("\n\nYou selected choice #2.\n");
          showmen();
          break;
        case '3':
          /* Choice #3 is to exit, so set a flag to end the main loop */
          terminate++;
          break;
        default:
          printf("\n\nInvalid selection.\n");
          showmen();
          break;
      }
    }
    /*************************************************************************/
    /* This is where the processing should be placed that is to occur while */
    /* awaiting user input. This program simply increments the variable     */
    /* "proc" to demonstrate that processing is not halted while awaiting   */
    /* input from standard input.                                           */
    /*************************************************************************/
    proc++;
  }
  /* We have left the main loop. Clean up and exit. */
  printf("\n\nYou have selected exit\n");
  printf("The proc variable was incremented %d times.\n",proc);
  printf("** Note**\n");
  printf("The increments took place while you were interacting with the menu.\n");
  printf("This demonstrates processing was not halted while waiting user input.\n");
}
```

In the above example, the statement to read the user input from the standard input device (STDIN) is placed within a "while" loop, followed by the code that should not be suspended.

The **read** statement is placed in an "if" clause. The return value from **read** determines what action takes place. If there is valid input from STDIN, the appropriate "case" statement is executed. If no information is available, a -1 is returned and the "proc++" statement is executed.

Conclusion

The number of times that the *proc* variable is incremented in the example should be substantially higher than the sum of the number of times that options #1 and #2 are selected. This demonstrates that the *proc* variable is incremented during the time that the interactive menu is presented.

If the value of the *proc* variable displayed is equal to the sum of the number of times that options #1 and #2 are selected, the most likely cause is that a version of MiniMON29K prior to Release 3.0 is being used on either the host or target system.

Suggested Reference

For more information, see the *Host Interface Specification*.

Appendix C

Defining a Trap to Switch to Supervisor Mode

This appendix describes how to place a 29K Family microprocessor or microcontroller in supervisor mode.

To switch a 29K Family processor to supervisor mode, the SM bit in the Current Processor Status (CPS) register must be set. But because the CPS register is a protected special-purpose register, it may not be modified when the processor is running in user mode.

However, when a trap is taken in a 29K Family processor, the processor is placed in supervisor mode. Because of this behavior, a user-defined trap can be used to switch the processor to supervisor mode.

Switching to Supervisor Mode

The **settrap()** host interface (HIF) service can be used to define a trap. At this point, you are in supervisor mode. However, you need the processor to remain in supervisor mode when returning from the trap.

Remaining in Supervisor Mode

When a trap is taken, a 29K Family processor copies the contents of the CPS register into the Old Processor Status (OPS) register. On return from this trap, the processor copies the contents of the OPS register into the CPS register.

So, if the user-defined trap sets the SM bit in the OPS register, the contents of OPS are copied into the CPS register on return from the trap. This keeps the processor in supervisor mode after returning from the trap.

Example Code

The following example consists of two files, one written in C (**trapit.c**) and the other written in 29K Family assembly language (**mytrap.s**). The file written in C contains the call to **settrap()** used to install the trap handler that sets the SM bit. Once the trap handler has been installed, an assembly language routine is called that will cause the newly installed trap to be asserted. Any code that is executed after this user-defined trap is asserted will execute in supervisor mode.

The 29K assembly language file contains the source code for the user-defined trap handler as well as a function that, when called, will cause the user defined trap to be asserted.

trapit.c

```
#include <stdio.h>
#include <hif.h>
extern void super_mode(void);
extern void as70(void);
void howdy() {
       int i;
       for(i=0;i<=10;i++) printf(”Hello!\n”);
}
main() {
       _settrap(70,&super_mode);
       as70();
howdy();
}
```

mytrap.s

```
       .global _super_mode
_super_mode:
       mfsr gr96, ops
       or gr96, gr96, 0x10
       mtsr ops, gr96
       iret
       .global _as70
_as70:
       asneq   70,gr96,gr96
       jmpi    lr0
       nop
```

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X