MiniMON29K™
Target Interface Process
MONTIP

MiniMON29K™ Target Interface Process: MONTIP, Release 3.0

The text pages of this document have been printed on recycled paper consisting of 50% recycled fiber and virgin fiber; the post-consumer waste content is 10%. These pages are recyclable.

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

Contents

About MONTIP

Chapter 1

Using MONTIP

Chapter 2

Using PCSERVER

Chapter 3

Initial Communications Between MONTIP and the Target

Chapter 4

MiniMON29K Message Communication System

Chapter 5

MiniMON29K Messages

Appendix A

MONTIP Error Messages

Appendix B

MiniMON29K Target Message System

Appendix C

Target Message Drivers

Index

Figures and Tables

Figures

Tables

About MONTIP

The Advanced Micro Devices (AMD®) MiniMON29K™ target interface process (TIP), **montip**, is the software application program that is invoked by a debugger front end (DFE) to communicate with a 29K™ Family target system running the MiniMON29K target-resident monitor software. **montip** conforms to AMD's Universal Debugger Interface (UDI) and can be used with UDI-compliant debugger front ends, such as: **mondfe**, which provides the MiniMON29K product's line-oriented user interface; **xray29u**, which provides the XRAY29K™ product's window-based user interface; or **gdb**, the GNU debugger.

This chapter first describes the features and modules of the **montip** software, then discusses the documentation associated with **montip**.

MONTIP Software

The features of the **montip** software are discussed below, followed by a description of the four modules of the program.

MONTIP Features

montip is the software application program that is invoked by a debugger front end (DFE) to communicate with a 29K Family target system running the MiniMON29K target-resident monitor software. **montip** conforms to AMD's Universal Debugger Interface (UDI) and can be used with UDI-compliant debugger front ends, such as: **mondfe**, which provides the MiniMON29K product's line-oriented user interface; **xray29u**, which provides the XRAY29K product's window-based user interface; or **gdb**, the GNU debugger. Figure 0–1 shows the relationship between **montip** and **mondfe**.

montip is the target interface process (TIP) for 29K Family-based target systems running the MiniMON29K software, and runs on a host computer system such as a PC or a Sun™ workstation. The communications interface between **montip** and the target is the MiniMON29K Message Communications Interface (see Chapter 4 for more information on the interface). This interface can be either a shared-memory interface of PC plug-in boards, or a serial communications link of a stand-alone board. In addition, for target systems with a parallel port, **montip** supports unidirectional parallel-port communications for downloading files from a PC.

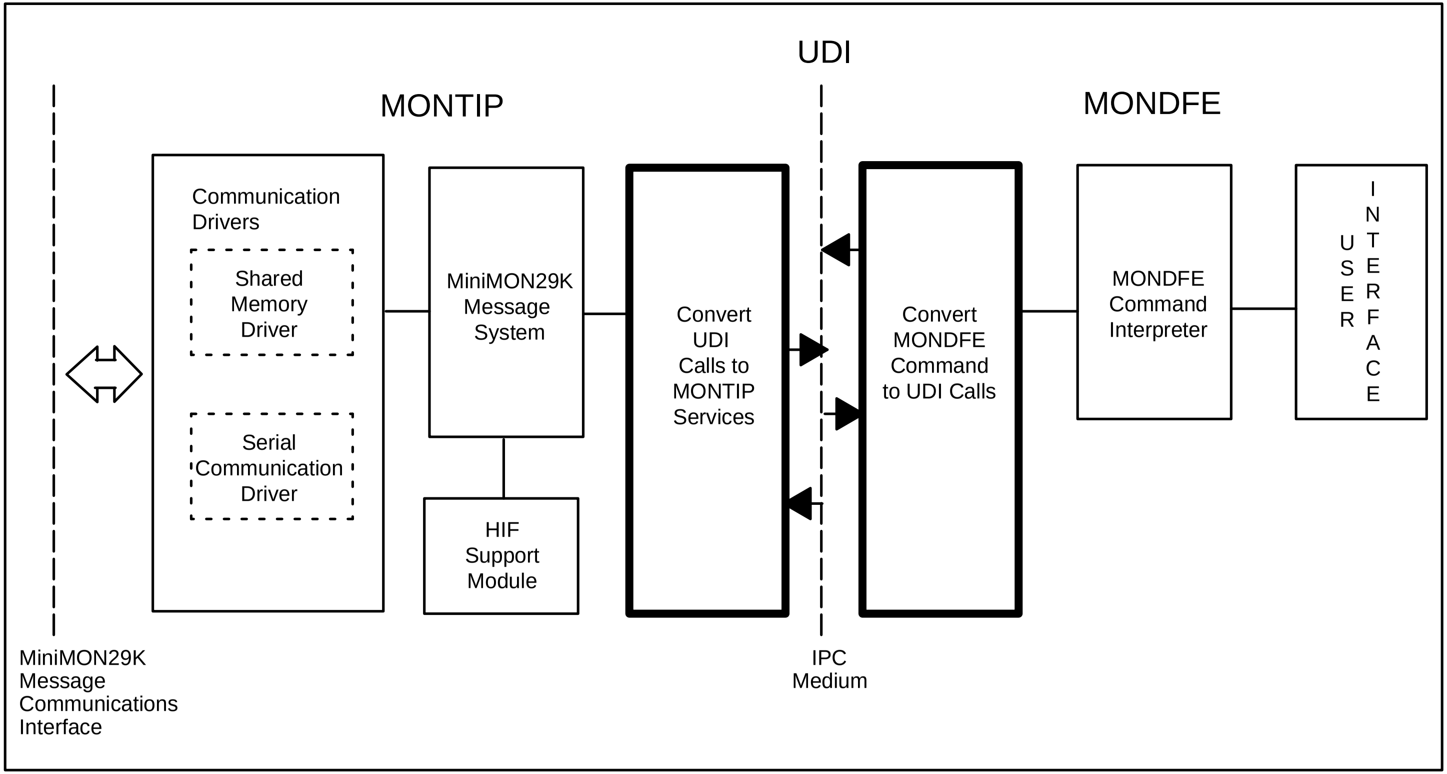

Figure 0–1. MiniMON29K MONTIP with UDI-Conformant DFE, MONDFE

The communications between **montip** and the application running on the target take place using MiniMON29K messages, which are structured streams of bytes. (Chapter 5 describes the structure and usage of the messages currently defined.) There are two types of messages:

- Debug messages. The debug messages are used by **montip** to communicate with the MiniMON29K monitor running on the target.
- OS messages. The OS messages are used by **montip** to communicate with the application or the operating system running on the target.

montip includes the serial communications drivers to send and receive messages for both MS-DOS and UNIX systems. **montip** also includes the communications drivers for the shared memory interface of the PC plug-in boards supported by AMD. The communications interface between **montip** and the target must be specified on the command line of **montip** at the time of invocation.

The MiniMON29K monitor software running on the target includes AMD's **osboot** and its host interface (HIF) kernel by default. The HIF kernel provides some of its services using **montip** running on an intelligent host computer system. **montip** includes the support routines for the HIF kernel of AMD's **osboot** running on the target. These routines are used to perform I/O operations on the host file system that are requested by the target application program.

MONTIP Modules

montip is made up of four modules, which are described on the following pages and illustrated in Figure 0–2.

NOTE: In this manual, "target" refers to the target system running the MiniMON29K monitor software—**osboot** and its HIF kernel, along with the debugger. "Host" refers to the system running **montip**.

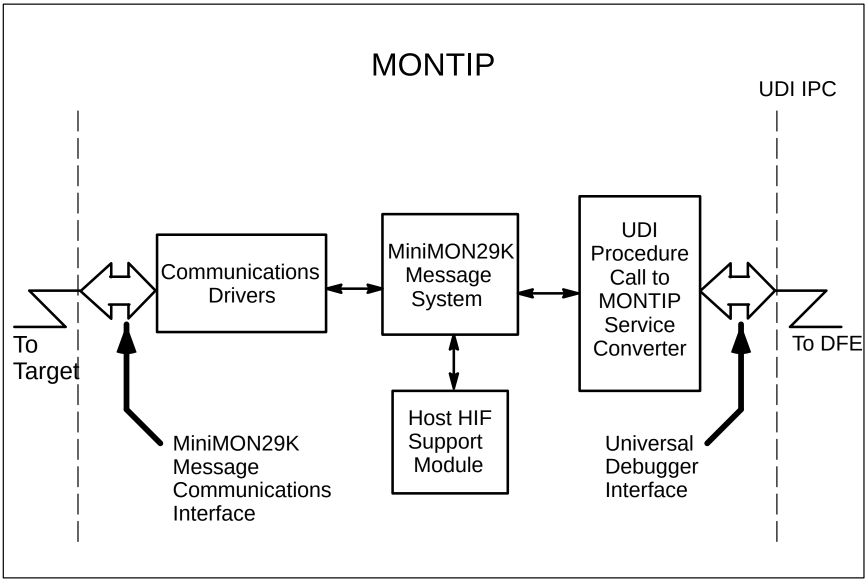

Figure 0–2. MiniMON29K Target Interface Process Modules

UDI Procedure Call to MONTIP Service Converter

This module implements the different UDI procedure calls using the services of the Message System module. It converts the UDI data structures to **montip** data structures and calls the Message System function to build the appropriate message. The module then sends the message to the 29K Family-based target running the MiniMON29K monitor software. Depending on the service requested, this module waits for the results. The results (if any) received from the target are put in UDI data structures and returned to the caller (debugger front end) through the UDI layer. The actual implementation of the transmission of the results depends on the UDI interprocess communication (IPC) mechanism used.

MiniMON29K Message System

This module implements the services to build, send, and receive MiniMON29K messages. It sends and receives messages using the communications handlers of the MiniMON29K Message Communications Interface. Every message has a message header followed by data, if applicable. The message header contains a message-code field and a message-length field. The different message codes and their corresponding message structures are defined in Chapter 5. When a MiniMON29K product message is received from the target system, **montip** examines the message-code field. If the message is one of the host interface (HIF) messages, **montip** invokes the Host HIF Support Module to service the message. Otherwise, **montip** saves the message in its receive buffer until a UDI procedure call requests it.

Host HIF Support

This module implements part of the run-time support provided by the HIF kernel of **osboot**. It is used to perform I/O operations on the host computer’s file system. The HIF kernel of **osboot** sends a HIF_CALL message to **montip**. This message is received and handled by this module and the results are sent back to the HIF kernel in a HIF_CALL_RTN response message. (See Chapter 5 for more information on the HIF_CALL and HIF_CALL_RTN messages.)

Communications Drivers

This module contains the drivers to transmit and receive character(s) through the MiniMON29K Message Communications Interface. It includes the serial communications handlers for MS-DOS® and UNIX® systems, and the shared-memory handlers for AMD’s 29K Family-based PC plug-in boards (such as the AMD EB29K™ or EB29030™ board).

MONTIP Documentation

This documentation is written for programmers using **montip** to develop applications based on a 29K Family target system running the MiniMON29K monitor, and for programmers customizing **montip**. For more information on these microprocessors and microcontrollers, see the list of suggested reference materials that follows.

About This Manual

Chapter 1: “Using MONTIP” describes how to invoke **montip** and provides command-line syntax and descriptions of all command-line options.

Chapter 2: “Using PCSERVER” describes the set up and use of **pcserver** to communicate between MiniMON29K software running a 29K Family-based PC plug-in board (such as the AMD EB29K™) and a remote **montip,** which uses MiniMON29K messages.

Chapter 3: “Initial Communications Between MONTIP and the Target” briefly describes the initial messages sent by **montip** and the target system to establish a synchronous connection.

Chapter 4: “MiniMON29K Message Communication System” describes how messages are sent. The message and driver layers of the system are described, as well as the communications interfaces supported.

Chapter 5: “MiniMON29K Messages” describes the messages used by **montip** to communicate with a target system running the MiniMON29K software.

Appendix A: “MONTIP Error Messages” describes the error messages reported by **montip** to the DFE.

Appendix B: “MiniMON29K Target Message System” describes the code for the target message system, contained in the **msg.s** file.

Appendix C: “Target Message Drivers” lists the filenames for the EB29K, EB29030™, EZ-030™, SA-29200™ and SA-29205™ target message drivers. The code for the SA-29200 and SA-29205 driver is listed.

Suggested Reference Material

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

- *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
- *Am29050™ Microprocessor User's Manual*
  Advanced Micro Devices, order number 14778
- *Am29050™ Data Sheet*
  Advanced Micro Devices, order number 15039
- *Am29200™ and Am29205™ RISC Microcontroller User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16362
- *Am29240™, Am29245™, and Am29243™ RISC Microcontrollers User's Manual and Data Sheet*
  Advanced Micro Devices, order number 17741
- *High C® 29K™ User's Manual*
  Advanced Micro Devices
- *High C® 29K™ Reference Manual*
  Advanced Micro Devices
- *Host Interface (HIF) Specification*
  Advanced Micro Devices, order number 11014
- *MiniMON29K™ User Interface: MONDFE*
  Advanced Micro Devices, order number 18442
- *Processor Initialization and Run-Time Services: OSBOOT*
  Advanced Micro Devices, order number 18275
- *Programming the 29K™ RISC Family*
  by Daniel Mann, P T R Prentice-Hall, Inc. 1994
- *RISC Design-Made-Easy Application Guide*
  Advanced Micro Devices, order number 16693
- *Universal Debugger Interface (UDI) Specification*
  Advanced Micro Devices, order number 18276

MONTIP Documentation Conventions

The Advanced Micro Devices manual *MiniMON29K Target Interface Process: MONTIP* uses the conventions shown in the following table (unless otherwise noted). These same conventions are used in all the 29K Family support product 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.

NOTE: In this manual, "target" refers to the target system running the MiniMON29K monitor software, which includes **osboot** and its HIF kernel, along with the debugger. "Host" refers to the system running **montip**.

Chapter 1

Using MONTIP

montip is the MiniMON29K target interface process (TIP) which conforms to the Universal Debugger Interface (UDI). It is the software application program that interfaces to 29K Family-based hardware platforms running the MiniMON29K target-resident monitor software.

montip is invoked by a UDI-compliant debugger front end (DFE) program, such as **mondfe**. Both the DFE and the TIP run on the host computer. The communication between **montip**, which is running on the host machine, and the target-resident monitor software running on the 29K Family-based hardware platform takes place using MiniMON29K product messages. These messages are streams of bytes which are interpreted by the message system that is included with **montip** and the target monitor software. Chapter 5 describes the structure and meanings of each of the various MiniMON29K product messages that are included with **montip** and the target monitor software.

The communications drivers for a shared-memory interface (for PC plug-in boards that are supported by AMD) and for serial communications are part of **montip**. The serial communications driver can support baud rates of up to 38400 bps on both MS-DOS and UNIX hosts (see page 5–2 to ensure reliable serial communications at higher baud rates).

montip can be used with UDI-compliant debugger front end (DFE) programs such as: **mondfe**, which provides the MiniMON29K product's line-oriented user interface; **xray29u**, which provides the XRAY29K product's window-based user interface; or **gdb**, the GNU debugger.

NOTE: See Chapter 2 if you want to run programs from a UNIX machine on a 29K Family PC plug-in board (such as the EB29030 or EB29K Execution Boards) located in a remote PC.

Invoking MONTIP

Syntax: montip –t *targetInterface* [–baud *baudRate*]
[–bl *blockLoopcount*] [–com *serialPort*] [–le]
[–m *messageFile*] [–mbuf *messageBufferSize*]
[–par *parallelPort*] [–port *portAddress*][–R | –P | –S]
[–r *romObjectFile*] [–re *retries*][–seg *segmentAddress*]
[–to *timeoutLoopcount*]

where:

–t *targetInterface*

Specifies the type of communications interface that exists between the host running **montip** and its target. The target is either a 29K Family stand-alone board running the MiniMON29K target-resident monitor software, or **pcserver** if debugging on a remote PC plug-in board. **montip** selects the appropriate communications driver based on the interface specified with this parameter (see Chapter 4 for more information on the drivers).

The value of *targetInterface* must be one of the following: **eb29k**, **eb030**, **lcb29k**, **yarcrev8**, **serial**, or **paral_1**. The first four values specify that the target interface is a shared memory interface and that it is similar to that of the EB29K, EB29030, YARC ATM™, or YARC Rev. 8 PC plug-in board, respectively.

When **serial** is specified, **montip** assumes that the communications interface uses a serial communications link. The desired baud rate at which message transmission should take place can be specified using the **–baud** option.

When running on an MS-DOS host, the target interface can be specified as **paral_1**. When **paral_1** is specified, **montip** uses a parallel port on the PC (**lpt1:** or **lpt2:**) to send messages to the target, and receives the response messages from the target through the serial port. Therefore, it requires the use of both a serial port and a parallel port on the MS-DOS host.

mondfe provides the **tip** command, which can be used to enable and disable the use of the parallel port by **montip**. When the parallel port is disabled, **montip** uses the serial port to send and receive messages. The **–par** option can be used to specify the parallel port to use (the default is **lpt1:**).

–baud *baudRate*

Specifies the baud rate to be used over the serial communications link. The default value of *baudRate* is 9600. (See page 5–2 for information on ensuring reliable serial communications at higher baud rates.)

–bl *blockLoopcount*

Specifies the loop count to decrement when waiting to receive an arbitrary number of bytes. The default value of *blockLoopcount* is 40000.

–com *serialPort*

Specifies the serial port to be used by **montip** for sending messages to, and receiving messages from, the target system. If the parallel port option (**–par**) is also specified on the command line, **montip** sends messages to the target using the specified parallel port and receives messages from the target using the specified serial port. For MS-DOS hosts, the valid values of *serialPort* are **com1:** and **com2:**. The default value of *serialPort* is **com1:** for MS-DOS hosts and **/dev/ttya** for UNIX hosts.

–le

Specifies that the orientation of the target system is little endian. Otherwise, **montip** assumes that the orientation of the target is big endian.

–m *messageFile*

Specifies the filename to be used to log the message transactions that occur between **montip** and the MiniMON29K target-resident monitor. If *messageFile* is not specified, no log file is created.

–mbuf *messageBufferSize*

Specifies the maximum size of a message to be used by **montip** when communicating with the target system. The value of *messageBufferSize* is ignored if it exceeds the maximum message buffer size allowed by the target message system.

–par *parallelPort*

Specifies the parallel port on the PC that **montip** should use to send messages to the target. The serial port option (**–com**) must also be specified when using this option, since **montip** receives the messages from the target through the serial port.

–port *portAddress*

Specifies the I/O-port base address of the PC plug-in board. The default value of *portAddress* is 208h. This option is ignored when the target interface is a serial communications link.

–R | –P | –S

Specifies the desired execution mode for the downloaded application programs when used with the AMD **osboot** host interface specification (HIF) kernel provided with the MiniMON29K product. The selected mode stays in effect for the entire debugging session. The **–R** option specifies physical mode, and **–P** specifies protected mode. The HIF kernel provided with the MiniMON29K monitor software implements protected mode by using a one-to-one mapping of physical addresses to virtual addresses using the Translation Look-Aside Buffer (TLB) registers. The **–S** option can be used to run application programs in supervisor mode with no translation. The default is protected mode (**–P**). In cases where the processor does not support protected mode, **–P** has no effect.

–r *romObjectFile*

Specifies the name of the common object file format (COFF) file, if any, to be downloaded into the hardware platform's writable ROM space. The COFF file is downloaded to the target system before it is reset. **montip** requires that the MiniMON29K target-resident monitor software, along with its message system, be downloaded and running on the target before debugging can take place. Therefore, this option must be specified when the target is a PC plug-in board.

The MiniMON29K target-resident monitor software provides debugging functions which are invoked by **montip** through MiniMON29K product messages. The drivers for serial communications and for the shared-memory interfaces of a PC plug-in board are included with **montip** and the MiniMON29K target-resident monitor software.

When the **–r** option is specified, **montip** checks the current working directory for the specified object file. If the object file is not found, **montip** searches the directories specified in the **path** environment variable by replacing the last directory with **lib**. For example, if the **path** environment variable is set to

```
c:\29k\bin;c:\29k\lib;d:\c600\bin;
```

then the directories **montip** searches for the target object to download are: **c:\29k\lib**, **c:\29k\lib**, and **d:\c600\lib** (in that order).

–re *retries* Specifies the number of retries to perform while sending a message to the target system. The default value of *retries* is 1000.

–seg *segmentAddress*
Specifies the address of the PC memory segment to be used by **montip** to access the PC plug-in board's memory. The default value of *segmentAddress* is D000h. This option is ignored when the target interface is a serial communications link.

–to *timeoutLoopcount*
Specifies the loop count to decrement before timing out while waiting to receive a message from the target system. The default value of *timeoutLoopcount* is 10000.

Files

udiconfs.txt UDI configuration file for MS-DOS hosts

udi_soc UDI configuration file for UNIX hosts

NOTE: If the appropriate UDI configuration file does not reside in the working directory of the debugger-front-end (DFE) program, an error message is posted. To use a configuration file in another directory, define the **UDICONF** environment variable by setting it to the full path of the UDI configuration file you want to use. After **UDICONF** is defined, the DFE program looks for the UDI configuration file in the path specified by **UDICONF**. If the file is not found, the program looks for it in the working directory.

Example

```
eb29k_id montip.exe –t eb29k –r eb29k.os
```

This entry in the **udiconfs.txt** file (for MS-DOS hosts) associates the TIP ID **eb29k_id** (first field) with **montip**, the MiniMON29K TIP. When **eb29k_id** is used as the TIP ID to a UDI-compliant DFE program (e.g., **mondfe**), **montip** is invoked and the string of options (**–t** and **–r**) is passed to **montip**. The **–t** option specifies that the target interface is similar to that of the EB29K Execution Board. The **–r** option specifies the filename of the object consisting of the MiniMON29K target-resident monitor software, message system, and the AMD **osboot** and HIF kernel. This common object file format (COFF) file is downloaded by **montip** before the target is reset.

Example

```
lcb29k_id montip.exe –t lcb29k –r lcb29k.os –port 2A0 –seg CC00
```

This entry in the **udiconfs.txt** file (for MS-DOS hosts) associates the TIP ID **lcb29k_id** (first field) with **montip**, the MiniMON29K TIP. When **lcb29k_id** is used as the TIP ID to a UDI-compliant DFE program (e.g., **mondfe**), **montip** is invoked and the string of options (**–t**, **–r**, **–port**, and **–seg**) is passed to **montip**. The **–t** option specifies that the target interface is similar to that of the YARC ATM PC plug-in board. The **–r** option specifies the filename of the object consisting of the MiniMON29K target-resident monitor software, message system, and the AMD **osboot** and HIF kernel. This common object file format (COFF) file is downloaded by **montip** before the target is reset. The **–port** option specifies the I/O-port base address to be used by **montip** to communicate with the PC plug-in board. The **–seg** option specifies the segment base address of the PC memory that should be used by **montip** to access the memory on the PC plug-in board.

Example

```
serial_id AF_UNIX sock384 montip –t serial –baud 38400
```

This entry in the **udi_soc** file (for UNIX hosts) associates the TIP ID **serial_id** (first field) with **montip**, the MiniMON29K TIP. When **serial_id** is used as the TIP ID to a UDI-compliant DFE program (e.g., **mondfe**), **montip** is invoked and the string of options (**–t** and **–baud**) is passed to **montip**. The **–t** option specifies that the target interface is a serial communications link. The **–baud** option specifies 38400 as the baud rate used by **montip** to communicate with the target. Since no **–com** option is given, the default serial port (**/dev/ttya**) will be used.

Chapter 2

Using PCSERVER

pcserver is a PC software application that lets you run programs written for an AMD 29K Family processor on a 29K Family PC plug-in board (such as the AMD EB29K or EB29030 Execution Boards) located in a remote PC from a UNIX machine. (**pcserver** is not necessary when running programs on a stand-alone board.) Once you have connected a null-modem cable from a serial port on the remote PC to a serial port on your UNIX host, **pcserver** uses MiniMON29K product messages to communicate with the target interface process (**montip**) software running on your UNIX host.

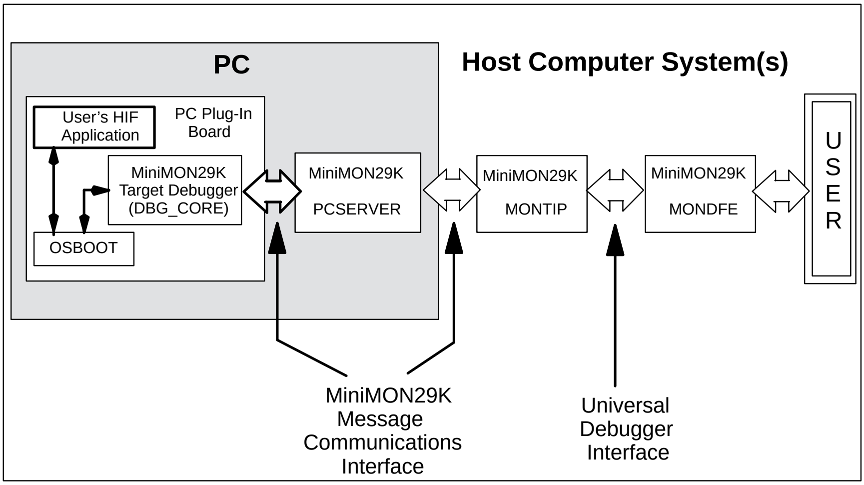

Figure 2–1.   Role of PCSERVER in the MiniMON29K Product

Invoking PCSERVER

Syntax: pcserver –r *romObjectFile* –t *targetInterface*
[–B *basePortAddress*] [–b *baudRate*][–M *messageRetries*]
[–m *messageFile*] [–p *serialPort*][–s *segmentAddress*]
[–T *timeout*] [–v]

where:

–r *romObjectFile*

Specifies the name of the common object file format (COFF) file to download into the PC plug-in board's writable ROM space. The COFF file is downloaded to the target system before it is reset. **pcserver** requires that the MiniMON29K target-resident monitor software be downloaded and running on the target before debugging can take place.

pcserver searches the working directory for the specified object file. If the object file is not found, **pcserver** searches the directories specified in the **path** environment variable by replacing the last directory with **lib**. For example, if the **path** environment variable is set to

```
c:\29k\bin;c:\29k\lib;d:\c600\bin;
```

then the directories **pcserver** searches for the target object to download are **c:\29k\lib**, **c:\29k\lib**, and **d:\c600\lib** (in that order).

–t *targetInterface*

Specifies the type of communications interface that exists between **pcserver** and the 29K Family hardware platform running the MiniMON29K target-resident monitor software. **pcserver** selects the appropriate communications driver based on the interface specified with this parameter.

The value of *targetInterface* must be one of the following: **eb29k**, **eb030**, **lcb29k**, or **yarcrev8**. These values specify that the target interface is a shared-memory interface and that it is similar to that of the AMD EB29K, AMD EB29030, YARC ATM, or YARC Rev. 8 PC plug-in boards, respectively.

–B *basePortAddress*

Specifies the I/O port base address of the PC plug-in board. The default value of *basePortAddress* is 208h.

–b *baudRate*
: Specifies the baud rate to be used over the serial communications link between the PC hosting the 29K Family PC plug-in board and the UNIX host running **montip**. The default value of *baudRate* is 9600.

–M *messageRetries*
: Specifies the number of retries to perform while sending a message to **montip** (running on the UNIX host). The default value of *messageRetries* is 1000.

–m *messageFile*
: Specifies the filename to be used to log the message transactions that occur between **pcserver** and the MiniMON29K target-resident monitor. If *messageFile* is not specified, no log file is created.

–p *serialPort*
: Specifies the serial port to be used by **pcserver** for communication with **montip** (running on the UNIX host). The valid values of *serialPort* are **com1:** and **com2:**. The default value is **com1:**.

–s *segmentAddress*
: Specifies the address of the PC memory segment to be used by **pcserver** to access the PC plug-in board's memory. The default value of *segmentAddress* is D000h.

–T *timeout*
: Specifies the loop count to decrement before timing out while waiting to receive a message from **montip** (running on the UNIX host). The default value of *timeout* is 10000.

–v
: Specifies verbose mode. In this mode, all of the messages are displayed on the screen.

Files

udiconfs.txt — UDI configuration file for MS-DOS hosts

udi_soc — UDI configuration file for UNIX hosts

NOTE: If the appropriate UDI configuration file does not reside in the working directory of the debugger-front-end (DFE) program, an error message is posted. To use a configuration file in another directory, define the **UDICONF** environment variable by setting it to the full path of the UDI configuration file you want to use. After **UDICONF** is defined, the DFE program looks for the UDI configuration file in the path specified by **UDICONF**. If the file is not found, the program looks for it in the working directory.

Example

```
pcserver –t eb29k –r eb29k.os –p com1: –b 9600
```

In the above example, the **–t** parameter specifies that the target interface is similar to that of the EB29K Execution Board. The **–r** parameter specifies the filename of the object consisting of the MiniMON29K target-resident monitor software, message system, and the AMD **osboot** and HIF kernel. This common object file format (COFF) file is downloaded by **montip** before the target is reset. The **–p** option specifies the serial port to use when receiving MiniMON29K product messages from **montip**.

NOTE: The two machines must be connected through a null-modem cable. The **–b** option specifies the baud rate to use for communications with **montip**.

Chapter 3

Initial Communications Between MONTIP and the Target

This chapter briefly describes the initial messages sent by **montip** and the target system to establish a synchronous connection.

The Message System module of **montip** communicates with its peer on the target system (see Figure 4–1 on page 4–2). They communicate using MiniMON29K messages, which are described in Chapter 5. The communications interface between the host running **montip** and the 29K Family-based target system running the MiniMON29K monitor software can be either a shared-memory interface of PC plug-in boards or a serial-communications link. The drivers to transmit and receive the messages across the communications interface are provided with the MiniMON29K product software (see Chapter 4 for more information on the drivers).

When the target system is powered up, the target sends a HALT message to the host. The HALT message is composed of six 32-bit words shown below in hexadecimal digits:

```
0000002B, 00000010, 00000005, <pc0_value>, <pc1_value>, 00000000
```

where *pc0_value* and *pc1_value* are the values of the Program Counter 0 (PC0) and Program Counter 1 (PC1) processor special-purpose registers.

NOTE: The messages would be followed by a 32-bit checksum of the message bytes when the communications interface is a serial communications link (see page 5–2 for more information).

When a debugger front end issues a connection request to **montip**, **montip** sends a CONFIG_REQ message to the target. The CONFIG_REQ message is composed of two 32-bit words shown below in hexadecimal digits:

```
00000001, 00000001
```

In response to the CONFIG_REQ message from **montip**, the target sends a CONFIG message to **montip**. On receipt of the CONFIG message, **montip** reports a successful connection to the debugger front end.

From this point on, **montip** services the UDI requests received from the debugger front end by sending appropriate message(s) to the target. The results received from the message responses from the target are sent back to the debugger front end. Thus, **montip** can operate with any UDI-conformant debugger front end.

Chapter 4

MiniMON29K Message Communication System

The message system of **montip** running on the host-computer system communicates with the message system of the monitor running on the 29K Family-based target system. The communications take place using MiniMON29K messages, which are structured streams of bytes. The MiniMON29K message protocol defines an acknowledgement message for every message, except for the initial message (HALT message) sent by the target system when powered up. After the message systems establish a synchronous connection (see Chapter 3), the target behaves as the message server by responding to the request messages received from **montip** with acknowledgement messages containing the results of the operation performed on the target.

A request–acknowledge message pair denotes one complete message transaction. The message system locks the communications channel until a transaction is completed. After the transaction is completed, the communications channel is freed for subsequent messages. This locking and freeing of the communications channel is done using a message semaphore. On the host system, the message system frees up the communications channel for subsequent messages after receiving the acknowledgement message from the target.

The message systems on the host and target use the communication drivers to physically send and receive the messages across the message communications interface. Figure 4–1 shows the MiniMON29K Message Communication System layers—the message layer and the driver layer. The message layer provides a device-independent interface to the communications interface. The driver layer implements the device-dependent routines to operate the communications device to send and receive messages. The driver layer may use the underlying operating-system services to read and write to the communications device.

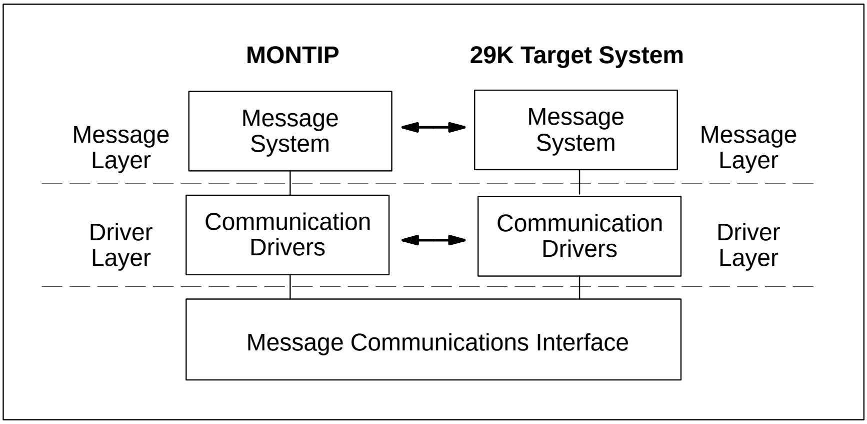

Figure 4–1. MiniMON29K Message Communication System Layers

The remainder of this chapter describes the components of the MiniMON29K message communication system:

- “Message Communications Interface” on page 4–3
- “MONTIP Message System” on page 4–5
- “MONTIP Message-Layer Interface” on page 4–11
- “MONTIP Drivers” on page 4–13
- “MiniMON29K Target Message System” on page 4–21
- “MiniMON29K Target Message-Layer Interface” on page 4–22
- “MiniMON29K Target Drivers” on page 4–26

NOTE: Throughout this chapter, “target” refers to the 29K Family-based target system running the MiniMON29K monitor software; “host” refers to the computer system running **montip**.

Message Communications Interface

The MiniMON29K target interface process, **montip**, runs on a host computer system, such as a PC or a Sun workstation. **montip** communicates with the 29K-Family target system running MiniMON29K monitor software using MiniMON29K messages, which are structured streams of bytes. The host and target support (and include drivers for) the following communications interfaces:

- Shared memory interface of a PC plug-in board

  In this type of interface, a data path exists between the PC host running **montip** and the PC plug-in board, which allows **montip** to access the memory on the PC plug-in board. Examples of PC plug-in boards hosting 29K Family microprocessors are: the AMD EB29K Execution Board, the AMD EB29030 Execution Board, the YARC Rev 8 board, and the YARC ATM (Sprinter) board.

- Serial communications interface of a stand-alone execution board

  In this type of interface, the serial port of the host running **montip** and the serial port of the stand-alone execution target system are connected via a serial cable. Examples of such systems are: the AMD SA-29200 Demonstration Board hosting the Am29200 microcontroller, the AMD SA-29205 Demonstration Board hosting the Am29205 microcontroller, and the AMD EZ-030 Demonstration Board hosting the Am29030 microprocessor.

- Parallel port interface between a PC and a stand-alone execution board (for MS-DOS hosts only)

  In this type of interface, the parallel port of the PC is connected to the parallel port on the stand-alone execution board via a parallel cable. Examples of such systems are: the AMD SA-29200 Expansion Board hosting the Am29200 or Am29205 microcontroller, and the AMD SA-29240™ board hosting the Am29240 microcontroller.

The drivers for the different communications interface are provided with the MiniMON29K product software. The communications drivers provide the functions to initialize the interface, and transmit and receive message byte streams. **montip** has built-in drivers for different communications interfaces and allows the user to select the appropriate drivers at the time of invocation. The target monitor includes only the drivers for the communications interface of that particular hardware system. For example, if the message communications interface is a serial communications link, then only the serial drivers are included in the target monitor. This helps keep the target monitor software small and excludes redundant software which could hinder debugging.

MONTIP Message System

The type of message communications interface that exists between the host computer system running **montip** and the 29K target system running the MiniMON29K monitor software is specified at the time of **montip** invocation. Based on the interface type specified, **montip** selects the low-level communications drivers from a table of entries that performs the necessary device operations to send and receive MiniMON29K messages.

The Message System module of **montip** defines a table of Target Driver Functions (TDF), using the following data structure:

```
typedef  struct target_dep_funcs {
 char    target_name[15];
 INT32   (*msg_send)(union msg_t *msg_buffer, INT32
         port_base);
 INT32   (*msg_recv)(union msg_t *msg_buffer, INT32
         port_base, INT32 mode);
 INT32   (*init_comm)(INT32 port_base, INT32 mem_seg);
 INT32   (*reset_comm)(INT32 port_base, INT32 mem_seg);
 INT32   (*exit_comm)(INT32 port_base, INT32 mem_seg);
 INT32   (*read_memory)(INT32 mspace, ADDR32 addr, BYTE
         *buf, INT32 count, INT32 port_base, INT32
         mem_seg);
 INT32   (*write_memory)(INT32 mspace, ADDR32 addr,
         BYTE *buf, INT32 count, INT32 port_base, INT32
         mem_seg);
 INT32   (*fill_memory)(void);
 INT32   PC_port_base;
 INT32   PC_mem_seg;
 void    (*go)(INT32 port_base, INT32 mem_seg);
} TDF;
```

The different elements of the above structure are explained below.

char target_name[15]

Contains a name that identifies the particular type of communications interface.

INT32 (*msg_send)()

Points to the function that sends the MiniMON29K message contained in the message buffer, **msg_buffer**. For shared memory interfaces, the **port_base** parameter contains the base address of the I/O port on the PC on which the 29K Family-based PC plug-in board is configured. This function call returns after the complete message is transmitted. The return value is a 0 (zero) if the message was successfully sent, and a –1 (minus 1) to indicate a failure.

INT32 (*msg_recv)()

Points to the function that polls the interface and reports the receipt of a new message from the target. The received message is stored in **msg_buffer**, which must be large enough to hold the incoming MiniMON29K message. The **mode** parameter specifies whether the polling should be blocking or nonblocking. In blocking mode, the function waits until a message is received. In nonblocking mode, the function times out waiting for a new message. For shared-memory interfaces, the **port_base** parameter contains the base address of the I/O port on the PC on which the 29K Family-based PC plug-in board is configured. This function returns a –1 (minus 1) if no message was received, and returns the message number if a valid message was received in the buffer.

INT32 (*init_comm)()

Points to the function that initializes the communications interface. For shared memory interfaces, **port_base** specifies the base address of the I/O port to control the board, and **mem_seg** specifies the segment address of the memory “window” on the PC host to use with that 29K Family-based PC plug-in board configuration.

INT32 (*reset_comm)()

Points to the function that resets the communications interface. For shared-memory interfaces, **port_base** specifies the base address of the I/O port to control the board, and **mem_seg** specifies the segment address of the memory “window” on the PC host to use with that 29K Family-based PC plug-in board configuration.

INT32 (*exit_comm)()

Points to the function that closes the communications interface. For shared-memory interfaces, **port_base** specifies the base address of the I/O port to control the board, and **mem_seg** specifies the segment address of the memory “window” on the PC host to use with that 29K Family-based PC plug-in board configuration.

INT32 (*read_memory)()

Points to the function that reads from the memory on the PC plug-in board hosting the 29K Family microprocessor. This function is valid only for shared-memory interfaces. This function reads **count** number of bytes from the 29K Family target memory space into the buffer pointed to by **BUF**. The origin for the read operation is specified by the memory space and offset specified by the **mspace** and **addr** parameters. The **port_base** parameter specifies the base address of the I/O port to control the board, and **mem_seg** specifies the segment address of the memory “window” on the PC host to use with that 29K Family-based PC plug-in board configuration. A 0 (zero) is returned if the read operation was successful; a –1 (minus 1) if unsuccessful.

INT32 (*write_memory)()

Points to the function that writes to the memory on the PC plug-in board hosting the 29K Family microprocessor. This function is valid only for shared-memory interfaces. This function writes **count** number of bytes from buffer, **buf**, to the offset and memory space specified by the **addr** and **mspace** parameters. The **port_base** parameter specifies the base address of the I/O port to control the board, and **mem_seg** specifies the segment address of the memory “window” on the PC host to use with that 29K Family-based PC plug-in board configuration. A 0 (zero) is returned if the write operation was successful; a –1 (minus 1) if unsuccessful.

INT32 (*fill_memory)()

Points to the function that fills, with a specified pattern, the memory on the PC plug-in board hosting the 29K Family microprocessor. It currently is not used by the message system.

INT32 PC_port_base

Contains the base address of the I/O port on the PC host on which the DIP switches on the 29K Family-based PC plug-in board is configured. This value is used only for shared- memory interfaces.

INT32 PC_mem_seg

Contains the segment address of the 16-Kbyte memory “window” on the PC on which the 29K Family-based PC plug-in board is configured. This value is used only for shared- memory interfaces.

void (*go)()

Points to the function that resets the 29K Family processor on the PC plug-in board. It is used only for shared-memory interfaces. The **port_base** parameter specifies the base address of the I/O port to control the board, and **mem_seg** specifies the segment address of the memory “window” on the PC host to use with that 29K Family-based PC plug-in board configuration.

A TDF array is initialized with the driver routines for the different message communications interfaces that are supported by **montip**. The TDF entries for the MS-DOS and UNIX systems are described on the following pages.

New communications interfaces can be added by adding an entry into the table of driver functions.

The string (first value) in each entry is the identifier to use on **montip**’s command-line at the time of invocation. **montip** selects the corresponding communications drivers from the TDF table defined.

Target Driver Functions (TDF) Array on MS-DOS Systems

On MS-DOS systems, **montip** uses the following table of entries for the target-driver functions array. These entries include the support routines for shared-memory interfaces of PC plug-in boards based on the 29K Family.

```
TDF TDF[] = {
”eb29030”, msg_send_eb030, msg_recv_eb030,
       init_comm_eb030, reset_comm_eb030,
       exit_comm_eb030, read_memory_eb030,
       write_memory_eb030, fill_memory_eb030, (INT32)
       0x208, (INT32) 0xd000, go_eb030,
”eb030”, msg_send_eb030, msg_recv_eb030,
       init_comm_eb030, reset_comm_eb030,
       exit_comm_eb030, read_memory_eb030,
       write_memory_eb030, fill_memory_eb030, (INT32)
       0x208, (INT32) 0xd000, go_eb030,
”eb29k”, msg_send_eb29k, msg_recv_eb29k,
       init_comm_eb29k, reset_comm_eb29k,
       exit_comm_eb29k, read_memory_eb29k,
       write_memory_eb29k, fill_memory_eb29k, (INT32)
       0x208, (INT32) 0xd000, go_eb29k,
”yarcrev8”, msg_send_eb29k, msg_recv_eb29k,
       init_comm_eb29k, reset_comm_eb29k,
       exit_comm_eb29k, read_memory_eb29k,
       write_memory_eb29k, fill_memory_eb29k, (INT32)
       0x208, (INT32) 0xd000, go_eb29k,
”lcb29k”, msg_send_lcb29k, msg_recv_lcb29k,
       init_comm_lcb29k, reset_comm_lcb29k,
       exit_comm_lcb29k, read_memory_lcb29k,
       write_memory_lcb29k, fill_memory_lcb29k,
       (INT32) 0x208, (INT32) 0xd000, go_lcb29k,
”paral_1”, msg_send_parport, msg_recv_serial,
       init_comm_serial, reset_comm_serial,
       exit_comm_serial, read_memory_serial,
       write_memory_serial, fill_memory_serial,
       (INT32) –1 , (INT32) –1, go_serial,
”serial”, msg_send_serial, msg_recv_serial,
       init_comm_serial, reset_comm_serial,
       exit_comm_serial, read_memory_serial,
       write_memory_serial, fill_memory_serial,
       (INT32) –1 , (INT32) –1, go_serial,
”\0”
};
```

The string (first value) in each entry shown identifies the communications interface as follows:

- “eb29030” or “eb030” specifies an interface similar to that of the EB29030 PC plug-in board.
- “eb29k” specifies an interface similar to that of the EB29K PC plug-in board.
- “yarcrev8” specifies an interface similar to that of the YARC Rev 8 PC plug-in board.
- “lcb29k” specifies an interface similar to that of the YARC ATM (Sprinter) PC plug-in board.
- “paral_1” specifies a unidirectional parallel communications interface for **montip** to send messages (data) to the target, and a serial interface for **montip** to receive messages (data) from the target.
- “serial” specifies a bidirectional serial communications interface between **montip** and the target.

Target Driver Functions (TDF) Array on UNIX Systems

In addition to the above entries, on UNIX systems, the target-driver-functions array also contains the entry shown below to communicate with MiniMON29K **pcserver** to execute programs on PC- hosted plug-in boards.

```
”pcserver”, msg_send_serial, msg_recv_serial,
       init_comm_serial, reset_comm_pcserver,
       exit_comm_serial, read_memory_serial,
       write_memory_serial, fill_memory_serial,
       (INT32) –1 , (INT32) –1, go_serial,
```

MONTIP Message-Layer Interface

The message layer defines two buffers to hold the incoming and outgoing messages:

```
union msg_t       *send_msg_buffer;
union msg_t       *recv_msg_buffer;
```

The variable **send_msg_buffer** points to the message buffer that is used to send messages to the target, and the variable **recv_msg_buffer** points to the message buffer that is used to receive messages from the target. These buffers are accessible to the driver layer also. The buffers are allocated by the **Mini_msg_init()** function, which initializes the message layer.

The message layer shown in Figure 4–1 on page 4–2 provides a device-independent procedural interface to operate the message communications interface. The message-layer functions index into the TDF array to call the appropriate low-level functions to perform the necessary operation. These functions are listed and described below.

INT32 Mini_msg_init(char *target_comm_name);

The message layer of **montip** should be initialized before using the message system. **Mini_msg_init()** allocates the message buffers to send and receive messages. Based on the **target_comm_name** string, it calls the driver function from the TDF table to initialize the communications interface. A 0 (zero) is returned on successful initialization, and a –1 (minus 1) is returned to indicate failure.

INT32 Mini_msg_exit(void);

Mini_msg_exit() closes the communication device and deallocates the message buffers. A return value of 0 (zero) indicates successful completion, and a –1 (minus 1) indicates failure.

INT32 Mini_msg_send(void);

Mini_msg_send() sends the message contained in the send buffer, **send_msg_buffer**, to the target. This calls the driver function to transmit the message bytes to the target. For shared-memory interfaces, the driver-layer function copies the message to the target memory on the PC plug-in board, and interrupts the target message system. For serial interface and parallel interface, the driver-layer routines transmit the message one byte at a time to the target, and return after the entire message is transmitted to the target. A 0 (zero) is returned to indicate successful transmission of the message, and a –1 (minus 1) is returned to indicate failure.

INT32 Mini_msg_recv(INT32 RecvMode);

Mini_msg_recv() returns a –1 (minus 1) if no new message was received into the receive buffer, **recv_msg_buffer**. When a new message is received, the MiniMON29K message code is returned to the caller. The **RecvMode** parameter can be either **BLOCK**, to indicate to wait until a message is received, or **NONBLOCK** to indicate to return if a message is not received. **Mini_msg_recv()** calls the driver-layer function, which handles the incoming message bytes. For shared- memory interfaces, the driver layer polls the mailbox address for a message interrupt from the PC plug-in board. When a message interrupt is posted, the message is read into the receive buffer, **recv_msg_buffer**, from the target system memory. For serial interfaces on MS-DOS systems, each incoming message byte interrupts **montip**. The interrupt handler gets the incoming byte from the device and stores it in **recv_msg_buffer**. For serial interfaces on UNIX systems, the driver function uses the **read()** system call to receive the incoming bytes. The bytes received are stored in **recv_msg_buffer**.

INT32 Mini_init_comm(void);

Mini_init_comm() initializes the communications interface. Based on the type of interface specified, the appropriate driver function from the TDF table is invoked.

INT32 Mini_reset_comm(void);

Mini_reset_comm() resets the communications interface and clears the message buffers. Based on the type of interface specified, the appropriate driver function from the TDF table is called.

INT32 Mini_exit_comm(void);

Mini_exit_comm() closes the communications interface. Based on the type of interface specified, the appropriate driver function from the TDF table is called.

INT32 Mini_go_target(void);

Mini_go_target() puts the 29K Family microprocessor on the target system in Reset mode by asserting the RESET input signal. This is valid only for shared-memory interfaces, when **montip** downloads the ROM monitor onto the target and asserts the RESET input signal to execute the ROM monitor.

MONTIP Drivers

The driver functions that operate the communications device interfaces implemented in **montip** are described in the sections that follow.

MONTIP Shared-Memory Interface Drivers

The interface between the PC host running **montip** and the PC plug-in board hosting the 29K Family microprocessor is a shared-memory interface. The interface provides some byte-wide I/O port registers and a 16-Kbyte “window” of memory, which is shared by both the PC host and the PC plug-in board. The base address (start address) of the I/O port registers can be configured with the DIP switches on the PC plug-in board. The segment address of the memory “window” on the PC host also can be specified with the DIP switches on the PC plug-in board. The 16-Kbyte memory “window” can be made to address the memory on the PC plug-in board by programming the I/O port registers, thus providing a data path between the host and the target. A bidirectional communication path is provided by the I/O port register called the “mailbox” register. The “mailbox” register is used by the host to interrupt the target and vice versa, unless the interrupts are masked on the board with the DIP switches.

Refer to the hardware reference manual of the PC plug-in board for more information on DIP switches and their uses. For MiniMON29K software, the DIP switches must be set to enable interrupts from the PC host to the target board, and to disable the interrupts from the target to the PC host.

montip provides the driver routines for the following 29K Family-based PC plug-in boards. The drivers for the AMD boards (the EB29K and the EB29030 board) are described in more detail on the following pages.

- AMD's EB29K board
- AMD's EB29030 board
- YARC's Rev 8 board
- YARC's ATM (Sprinter) board

The I/O port base address and the segment address of the memory "window" to use can be specified on the command line of **montip** at the time of invocation using the **–port** and the **–seg** options.

The EB29K and EB29030 Interface Drivers

The interface between the PC host running **montip** and AMD's EB29K and EB29030 boards are quite similar. The target-driver functions for the EB29K interface and the EB29030 interface are listed in the TDF array for "eb29k" and "eb29030" target communications types, respectively (see page 4–9).

The EB29K and EB29030 boards running the MiniMON29K monitor software are controlled from the PC host running **montip** through four byte-wide I/O ports and a 16-Kbyte shared-memory "window." The I/O ports start sequentially at offset 0 from the base address specified when the board is configured with DIP switches. At offset 0h from the I/O port base is the control-port register. The control-port register is used to send control signals from the PC to the target board. The segment address of the 16-Kbyte shared-memory window is set by writing to the control-port register. At offset 1h and 2h from the I/O port base are two address registers. The address registers are used to set the base address of the 16-Kbyte memory "window" on the target which is mapped to the segment address of the memory "window" on the PC host. Thus by accessing the shared-memory window on the PC host, **montip** can access any memory location on the target board. At offset 3h from the I/O port base is the "mailbox" register. The "mailbox" register is mapped to offset 80800000h in the EB29K address space, and is mapped to offset 90000000h in the EB29030 address space. **montip** writes to the "mailbox" register to generate an interrupt on the target.

The driver routines for the EB29K and EB29030 boards are described below.

```
INT32 msg_send_eb29k(union msg_t *msg_ptr, INT32 port_base)
INT32 msg_send_eb030(union msg_t *msg_ptr, INT32 port_base)
```

The **msg_send_eb29k()** and **msg_send_eb030()** functions send a MiniMON29K message to the target, and interrupt the target execution. The message contained in **msg_ptr** is copied to the receive buffer on the target memory space. The receive buffer of the target monitor is at offset 80000404h in the EB29K address space, and at offset 404h in the EB29030 address space. After copying the message onto the target receive buffer, **montip** interrupts the target by writing to the "mailbox" register. The **port_base** parameter specifies the I/O port base address on the PC host. A 0 (zero) is returned for successful completion, otherwise a –1 (minus 1) is returned.

```
INT32 msg_recv_eb29k(union msg_t *msg_ptr, INT32 port_base, INT32 Mode)
INT32 msg_recv_eb030(union msg_t *msg_ptr, INT32 port_base, INT32 Mode)
```

The **msg_recv_eb29k()** and **msg_recv_eb030()** functions poll the "mailbox" register for new incoming messages. The monitor running on the target writes FFh to the "mailbox" register to indicate a new message in the buffer. The target also stores the pointer to where the message is on the target memory space—at offset 80000400h in the EB29K address space, and at offset 400h in the EB29030 address space. **msg_recv_eb29k()** and **msg_recv_eb030()** read the contents of the message from the target memory into the **msg_ptr** buffer. **montip** then writes FFh to the "mailbox" register to indicate receipt of the message, and resets to 0 (zero) the content of offset 80000400h in the EB29K memory space and offset 400h in EB29030 memory space. The **port_base** parameter specifies the I/O port base address on the PC host. The **Mode** parameter is not used. The message code of the new message received is returned, otherwise a –1 (minus 1) is returned to indicate no new message in the buffer.

```
INT32 init_comm_eb29k(INT32 port_base, INT32 mem_seg)
INT32 init_comm_eb030(INT32 port_base, INT32 mem_seg)
```

The **init_comm_eb29k()** and **init_comm_eb030()** functions write to the control-port register to set the base address of the memory window on the PC host to **mem_seg**. The functions also write to the address registers to set the corresponding memory window to offset 0h in the target memory space. These functions set the control bit to enable interrupts from the PC host to the target. The **port_base** parameter specifies the I/O-port base address on the PC host. A 0 (zero) is returned for successful completion; otherwise, a –1 (minus 1) is returned.

```
INT32 reset_comm_eb29k(INT32 port_base, INT32 mem_seg)
INT32 reset_comm_eb030(INT32 port_base, INT32 mem_seg)
```

The **reset_comm_eb29k()** and **reset_comm_eb030()** functions are the same as the **init_comm_eb29k()** and **init_comm_eb030()** functions, respectively.

```
INT32 exit_comm_eb29k(INT32 port_base, INT32 mem_seg)
INT32 exit_comm_eb030(INT32 port_base, INT32 mem_seg)
```

The **exit_comm_eb29k()** and **exit_comm_eb030()** functions are defined as empty functions that always return a 0 (zero).

```
INT32 read_memory_eb29k(INT32 mspace, ADDR32 addr, BYTE *data,
               INT32 count, INT32 port_base, INT32 mem_seg)
INT32 read_memory_eb030(INT32 mspace, ADDR32 addr, BYTE *data,
               INT32 count, INT32 port_base, INT32 mem_seg)
```

The **read_memory_eb29k()** and **read_memory_eb030()** functions program the address control registers with the offset specified in **addr**. This positions the 16-Kbyte memory "window" in the target address space from where **count** bytes of data from the memory on the target board are read into the **data** buffer in the PC host memory space. A 0 (zero) is returned if the read was performed successfully; otherwise, a –1 (minus 1) is returned.

```
INT32 write_memory_eb29k(INT32 mspace, ADDR32 addr, BYTE *data,
               INT32 count, INT32 port_base, INT32 mem_seg)
INT32 write_memory_eb030(INT32 mspace, ADDR32 addr, BYTE *data,
               INT32 count, INT32 port_base, INT32 mem_seg)
```

The **write_memory_eb29k()** and **write_memory_eb030()** functions program the address control registers with the offset specified in **addr**. This positions the 16-Kbyte memory "window" in the target address space where **count** bytes from the **data** buffer are copied from the PC host memory. A 0 (zero) is returned if the write was performed successfully, otherwise a –1 (minus 1) is returned.

```
void go_eb29k(INT32 port_base, INT32 mem_seg)
void go_eb030(INT32 port_base, INT32 mem_seg)
```

The **go_eb29k()** and **go_eb030()** functions toggle the RESET bit in the control-port register. Writing a 1 (one) to the reset bit in the control register resets the 29K Family microprocessor and starts execution.

```
INT 32 fill_memory_eb29k(void)
INT 32 fill_memory_eb030(void)
```

The **fill_memory_eb29k()** and **fill_memory_eb030()** functions are defined as empty functions that always return a 0 (zero).

The YARC Rev 8 and YARC ATM Interface Drivers

The target-driver functions for the YARC Rev 8 and the YARC ATM interface are listed in the TDF array for “yarcrev8” and “lcb29k” target communications types, respectively (see page 4–9).

MONTIP Serial-Interface Driver

The communications between **montip** running on a host computer system and a stand-alone target execution board running the MiniMON29K monitor software is through a serial interface. The serial port on the host computer is connected to the serial port on the stand-alone execution board via a serial cable. The baud rate and the host serial port that **montip** should use for communications can be specified on the command line at the time of invoking **montip** using the **–baud** and **–com** options.

montip implements a simple serial interface with one stop bit, no parity, and 8 bits per byte. Every message is appended with a 32-bit checksum value, which is the sum of all the bytes in the message (see page 5–2 for more information on checksums). The receiver checks the checksum received with the checksum of the message bytes received before posting a valid message interrupt to the message system. If the received message is valid, then an ACK message is sent to the transmitter. If the received message is invalid, then a NACK message is sent to the transmitter. The ACK and NACK messages are handled by the communications driver routines. The receipt of an ACK message marks the completion of a message transaction.

The target-driver functions for the serial interface defined in the TDF array for the “serial” target communications type (see page 4–9) are described below.

```
INT32 msg_send_serial(union msg_t *msg_ptr, INT32 port_base)
```

The **msg_send_serial()** function is used to send the MiniMON29K message contained in **msg_ptr** to the target via the serial interface. The **port_base** parameter is ignored. The **msg_send_serial()** function computes the checksum for the message, which is the sum of all the bytes of the message. The function appends the checksum to the end of the message. Note that the **msg_ptr** buffer should be large enough to append a checksum at the end of the message. The message and its checksum are then transmitted to the target using the **send_bfr_serial()** function, which uses the underlying operating-system services to transmit the message bytes. After transmitting the message and the checksum, the **msg_send_serial()** function waits to receive an ACK message from the target to indicate successful transmission. If an ACK message is received, **msg_send_serial()** returns a 0 (zero) to indicate successful transmission of the message. If a NACK message is received, **msg_send_serial()** resets the serial interface and resends the message. The maximum number of attempts to resend the message is specified by the **–re** command-line option. A –1 (minus 1) is returned to indicate failure during transmission of the message.

```
INT32 msg_recv_serial(union msg_t *msg_ptr, INT32 port_base, INT32 Mode)
```

The **msg_recv_serial()** function is called to find out if a new message has arrived. It returns the message received in the **msg_ptr** buffer. The **Mode** parameter is set to either **BLOCK** or **NONBLOCK**. When **Mode** is set to **NONBLOCK**, **msg_recv_serial()** returns immediately if no new message has arrived. When **Mode** is set to **BLOCK**, **msg_recv_serial()** waits (blocks) until a message is received from the target. The length of the wait can be specified using the **–bl** command option. The **port_base** parameter is ignored.

The **msg_recv_serial()** function calls the **recv_bfr_serial()** function, which copies the received message bytes from the underlying operating-system buffer or the circular buffer (**serial_io_buffer** in MS-DOS hosts) to the message **msg_ptr** buffer. **msg_recv_serial()** returns if a complete message header (8 bytes) has not arrived.

From the message header received, the number of bytes to follow the header is determined. **msg_recv_serial()** then calls **recv_bfr_serial()** to receive the remaining bytes plus the 32-bit checksum value. The checksum of the received message is computed and compared with the checksum value received from the target. If the checksums are equal, an ACK message is sent to the target, and **msg_recv_serial()** returns the message code of the received message. If the checksums are not equal, a NACK message is sent to the target, and a –1 (minus 1) is returned to the caller to indicate failure while receiving a message from the target.

```
INT32 init_comm_serial(INT32 port_base, INT32 mem_seg)
```

This function initializes the serial interface depending on the host used and the options given to **montip** on the command line.

On MS-DOS hosts, the **init_comm_serial()** function uses the BIOS services to initialize the serial interface. Based on the I/O port specified to the **–com** option, the I/O-port base address and the interrupt line for the serial communications controller SCC8259 are determined. The serial port is initialized for the baud rate specified on the **montip** command line. The transmit interrupt is disabled such that **montip** uses a polling loop while transmitting message bytes to the target. The receive interrupt of the serial port is enabled to generate an interrupt for every incoming byte from the target. The **init_comm_serial()** function installs the serial port interrupt handler, **serial_int()**, to handle the receive interrupts.

```
void interrupt serial_int()
```

The **serial_int()** interrupt handler buffers the incoming characters into a circular buffer and raises a flag if the buffer overflows. The circular buffer, **serial_io_buffer**, is initialized by the **init_comm_serial()** function. It returns a 0 (zero) to indicate successful completion, and a –1 (minus 1) to indicate a failure termination.

On UNIX hosts, the **init_comm_serial()** function opens the serial port specified by the **–com** command-line option for reading and writing using the **open()** system call. The serial-port parameters such as the baud rate, character size, parity, and number of stop bits are set using the **ioctl()** system call. The serial port is configured to perform nonblocking read and write operations. The serial port input and output buffers are flushed to discard their previous contents. It returns a 0 (zero) to indicate successful completion, and a –1 (minus 1) to indicate failure termination.

```
INT32 reset_comm_serial(INT32 port_base, INT32 mem_seg)
```

On MS-DOS hosts, this function resets the circular buffer and discards the previous contents of the buffer. This function clears any communications errors that might have occurred and any receive interrupts that are pending to be handled. A 0 (zero) is returned to indicate successful completion, and a –1 (minus 1) to indicate failure.

On UNIX hosts, this function resets the input and output buffers of the serial port using the **ioctl()** system call. It returns a 0 (zero) to indicate successful completion, and a –1 (minus 1) to indicate failure termination.

```
INT32 exit_comm_serial(INT32 port_base, INT32 mem_seg)
```

On MS-DOS hosts, this function resets the circular buffer, and installs the original vector corresponding to the serial port. It returns a 0 (zero) to indicate successful completion, and a –1 (minus 1) to indicate failure termination.

On UNIX hosts, this function resets the input and output buffers of the serial port and closes the serial port using the **close()** system call. It returns a 0 (zero) to indicate successful completion, and a –1 (minus 1) to indicate failure termination.

```
read_memory_serial()
write_memory_serial()
fill_memory_serial()
```

The functions **write_memory_serial()**, **read_memory_serial()**, and **fill_memory_serial()** are defined as empty functions and always return a –1 (minus 1).

```
void go_serial(INT32 port_base, INT32 mem_seg)
```

This function is an empty function and returns immediately.

MONTIP Parallel-Port Interface Driver

The parallel port interface is available for the PC only and is unidirectional (messages are sent to the target through the parallel port and received from the target through the serial port). Thus, the functions are the same as those described for serial communications. The exception is the replacement of the **msg_send_serial** function with the **msg_send_parport** function.

MiniMON29K Target Message System

The MiniMON29K message system on the target is the message server for the debugger and the application program running on the target system. It sends messages to and receives message from **montip** running on the host computer system. The target message system and its communications drivers are coded in 29K assembly language. The functions do not require any processor registers to be reserved for their use, and they execute in their own address space. The message layer provides a device-independent interface to the message communications interface. The driver layer implements the device-dependent functions to send and receive message bytes across the message communications interface. The message-layer functions and the driver functions are written according to the AMD calling conventions.

The MiniMON29K message-layer functions are the same for all target systems. The MiniMON29K product software includes the drivers for AMD-supported 29K Family-based target systems to send and receive messages to and from **montip** running on the host. The drivers included with the MiniMON29K product are:

- Shared-memory interface drivers for AMD's EB29K, AMD's EB29030, YARC's Rev 8, and YARC's ATM (Sprinter) PC plug-in boards.
- Serial communications drivers for SCC8530 device on AMD's EZ-030 stand-alone execution board, for the Am29200 on-chip serial port on AMD's SA-29200 stand-alone execution board, and for the Am29205 on-chip serial port on AMD's SA-29205 stand-alone execution board.
- Parallel-port driver to receive messages through the Am29200 or Am29205 on-chip parallel port on AMD's SA-29200 expansion board, and through the Am29240 on-chip parallel port on AMD's SA-29240 board.

The appropriate drivers to support the communications interface between the target and the host are linked together with the rest of the monitor software. The monitor is either downloaded to the target memory as in the case of PC plug-in boards, or is programmed in EPROMs on the stand-alone execution boards.

MiniMON29K Target Message-Layer Interface

The message layer defines a buffer, **_msg_rbuf**, to hold the incoming messages from **montip**, and a pointer, **_msg_next_p**, which gives the location where the next received character is to be stored:

```
         .global  _msg_rbuf
_msg_rbuf:        .block MSG_RBUF_SIZE
         .global  _msg_next_p
_msg_next_p:      .block 4
```

MSG_RBUF_SIZE gives the maximum size of the message buffer. It is the responsibility of the host to send messages no larger than **MSG_RBUF_SIZE** bytes. **_msg_next_p** is updated by the driver routines as the received message bytes are stored into **_msg_rbuf**. **_msg_next_p** is initialized to **_msg_rbuf** during reset and after the completion of a message transaction. **_msg_next_p** and **_msg_rbuf** are global variables and are accessible to the driver-layer functions.

The message layer also defines a global pointer, **_msg_sbuf_p**, which points to the location of current message that should be sent to the host before the next message is sent:

```
         .global  _msg_sbuf_p
_msg_sbuf_p:      .block 4
```

The **_msg_sbuf_p** pointer is reset to 0 (zero) after the message has been successfully transmitted to the host. Thus, **_msg_sbuf_p** is used as a semaphore to indicate that the message communications channel is busy when **_msg_sbuf_p** is a nonzero value, or free when **_msg_sbuf_p** is zero.

The message layer provides the following device-independent procedural interface to the message communications interface. These functions are called by the debugger and the operating system/application program running on the target system, and not by the driver-layer functions.

void msg_init(void)

This function initializes the message layer, and calls the driver initialization routine, **msg_initcomm**, to initialize the communications interface. It sets **_msg_next_p** to point to **_msg_rbuf** and clears the **_msg_sbuf_p** semaphore. The **msg_init()** function must be called before using the message system. The bootstrap code is required to install the necessary interrupt vectors before calling the **msg_init()** function.

int msg_send(msg_t *msg_buf);

This function sends the message contained in the buffer pointed to by **msg_buf** to the host. Before calling the driver function to send the message, it determines whether the message channel is free by examining the **_msg_sbuf_p** variable. If **_msg_sbuf_p** is zero and the message channel is free, **msg_send()** locks the message channel by writing the address of **msg_buf** to **_msg_sbuf_p**. It then calls the driver function to write out the message bytes through the message communications interface to the host. The driver function to write the message bytes is accessed through an indirect pointer, **msg_write_p**, as shown below:

```
.extern msg_write_p   ; pointer to driver write function
const  gr96, msg_write_p
const  gr96, msg_write_p
load   0, 0, gr96, gr96  ; get msg_write driver function
calli  lr0, gr96       ; call the driver function
nop
```

The **gr96** and **lr0** registers are saved before calling the driver function and restored on return from the driver function. The **msg_write_p** pointer is initialized by the driver initialization routine, **msg_initcomm**, with the write routine defined by the driver layer for that communications interface.

msg_send() returns a 0 (zero) if the message was sent successfully. It returns a –1 (minus 1) to indicate failure to send the message either due to transmission error or due to a lock on the **_msg_sbuf_p** semaphore.

int msg_wait_for(void);

This function is used to determine if the receive buffer contains a valid message from the host that needs to be processed. It returns a –1 (minus 1) to indicate that the receive buffer contains a valid message, and returns a 0 (zero) to indicate that no new message is in the receive buffer. It calls the driver function using the function pointer, **msg_wait_for_p**, as shown below:

```
.extern  msg_wait_for_p       ; pointer to driver
                              ; msg_wait_for function
const  gr96, msg_wait_for_p
consth gr96, msg_wait_for_p
load   0, 0, gr96, gr96       ; get function address
calli  lr0, gr96              ; call driver function
nop
```

The **lr0** register is saved before calling the driver function and is restored on return from the driver function. The **msg_wait_for_p** pointer is initialized by the driver initialization routine, **msg_initcomm**, with the wait-for-message routine defined by the driver layer for that communications interface.

When the communications interface is driven in interrupt mode, the driver-layer function returns immediately with a return value of 0 (zero) to indicate no new message has arrived. When the communications interface is driven in polled mode, the driver-layer function returns only when a valid message is received in the receive buffer, and returns a value of –1 (minus 1). The message layer calls the driver-layer function independent of whether the interface is in polled mode or in interrupt mode.

msg_V_arrive

The message layer provides an entry point for the driver layer to notify when a message has been received from the host. The driver-receive interrupt handler posts a message interrupt to the message system by jumping to the label, **msg_V_arrive**, inside the message system when a complete message is received in the receive buffer, **_msg_rbuf**. **msg_V_arrive** is defined as a virtual interrupt handler. It determines whether the message contained in **_msg_rbuf** is a MiniMON29K debug message or an operating-system message. If a debug message is received, it posts an interrupt to the debugger by jumping to the **dbg_V_msg** label inside the MiniMON29K debugger. If an operating-system message is received, it posts an interrupt to the operating system on the target by jumping to the **os_V_msg** label inside the operating system. The **msg_V_arrive** message interrupt handler is as shown below:

```
       .global msg_V_arrive
msg_V_arrive:
       const   gr4, _msg_rbuf
       consth  gr4, _msg_rbuf
       load    0, 0, gr4, gr4 ; determine message
code
       cpgeu   gr4, gr4, 64   ; is it an OS message
       jmpt    gr4, os_msg    ; yes, go to os_msg
       const   gr4, dbg_V_msg ; else
       consth  gr4, dbg_V_msg ; interrupt debugger
       jmpi    gr4            ; at dbg_V_msg
       nop
os_msg:
       const   gr4, os_V_msg  ; interrupt OS
       consth  gr4, os_V_msg  ; at os_V_msg
       jmpi    gr4
       nop
```

MiniMON29K Target Drivers

The driver functions to operate the communications device interface for the specific target hardware system are linked together with the message-layer module. For each type of communications interface, the driver layer must define a write function to send the message to **montip**, define a message-wait-for function to receive a message from **montip** (in polled mode), and define an interrupt handler to handle message interrupts from the host.

For each target hardware system, the **msg_initcomm** driver initialization function must be defined. **msg_initcomm** is called from the **msg_init()** function in the message layer. The **msg_initcomm** function should initialize the **msg_write_p** and **msg_wait_for_p** function pointers with the appropriate routines for the communications interface applicable to that target hardware system.

Target Shared-Memory Interface Drivers

The drivers for the shared-memory interface of the following PC plug-in boards are provided with the MiniMON29K product software. The drivers for the AMD boards (the EB29K and the EB29030 board) are described in more detail on the following pages.

- AMD’s EB29K board
- AMD’s EB29030 board
- YARC Rev 8 board
- YARC ATM (Sprinter) board

The EB29K and EB29030 Message Drivers

`ASM int msg_initcomm(void)`

ASM is used to denote that **msg_initcomm** is an assembly-level label, and has no leading underscore. The **msg_initcomm** function is called from the message-layer initialization function, **msg_init()**. The interrupt handler, **msg_intr**, for the interrupt line used by the communications interface must be installed during the bootstrap process.

The **msg_initcomm** function reads the "mailbox" register clearing any pending interrupts. The **msg_write_p** and **msg_wait_for_p** pointers are then initialized with the board-specific **write** and **msg_wait_for** functions, which write out a message and wait for a message, respectively. The **msg_initcomm** function returns the driver version number in gr96 to the caller.

The code below shows the **msg_initcomm** function for AMD's EB29030 board. **msg_eb030_write** and **msg_eb030_wait_for** are the functions to write a message and wait for a message for the EB29030 board, respectively. The **msg_initcomm** function for the EB29K board is similar and installs the **msg_eb29k_write** and **msg_eb29k_wait_for** functions instead. The "mailbox" register is at offset 90000000h for the EB29030 board, and is at offset 80800000h for the EB29K board.

```
;
----------------------------------------MSG_INITCOMM
; return version number in gr96.
    .equ COMM_VERSION,0x06
    .equ mailbox,0x90000000
    .extern msg_write_p
    .extern msg_wait_for_p
msg_initcomm:
    const   gr96, mailbox
    consth  gr96, mailbox
    load    0, 0, gr96, gr96     ; clear mail box

    const   gr96, save_regs
    consth  gr96, save_regs
    store   0, 0, gr97, gr96     ;backup gr97

    const   gr96, msg_write_p
    consth  gr96, msg_write_p
    const   gr97, msg_eb030_write
    consth  gr97, msg_eb030_write
    store   0, 0, gr97, gr96     ; msg_write
```

```
    const   gr96, msg_wait_for_p
    consth  gr96, msg_wait_for_p
    const   gr97, msg_eb030_wait_for
    consth  gr97, msg_eb030_wait_for
    store   0, 0, gr97, gr96     ; msg_wait_for

    const   gr96, save_regs
    consth  gr96, save_regs
    load    0, 0, gr97, gr96     ; restore gr97

    jmpi    lr0
    const   gr96, COMM_VERSION
```

```
ASM void msg_eb29k_write(void)
ASM void msg_eb030_write(void)
```

ASM is used to denote that these labels are assembly-level labels, and have no leading underscore. The **msg_eb29k_write** and **msg_eb030_write** functions are called from the **msg_send()** function. For shared-memory interfaces, **msg_send()** writes a pointer, to the location of the message on the target address space, into the **_msg_sbuf_p** semaphore, before calling the driver write function. The driver-layer write function posts the message to **montip** running on the host by writing a –1 (FFh) to the “mailbox” register. This indicates to **montip** that a message is ready in the buffer. Note that the DIP switches on the board must be set such that writing to the “mailbox” register does not generate an interrupt on the PC. **montip** running on the PC host polls the “mailbox” register from the PC side until it reads a –1 (FFh), which indicates that a message is ready to be received.

The code below shows the driver-layer write function for the EB29030 board.

```
;
-------------------------------------MSG_EB030_WRITE
; write 0xff to mailbox, return.
.equ mailbox,0x90000000
msg_eb030_write:
    const   gr4, save_regs
    consth  gr4, save_regs
    store   0, 0, gr96, gr4      ; backup gr96
    const   gr96, save_regs+4
    consth  gr96, save_regs+4
    store   0, 0, gr97, gr96     ; backup gr97
```

```
    const  gr96, mailbox
    consth gr96, mailbox
    constn gr97, –1           ; write 0xff to mailbox
    store  0, 0, gr97, gr96    ; message ready in
                               ; buffer.

    const  gr96, save_regs+4
    consth gr96, save_regs+4
    load   0, 0, gr97, gr96    ; restore gr97
    const  gr96, save_regs
    consth gr96, save_regs
    load   0, 0, gr96, gr96    ; restore gr96

    jmpi   lr0
    nop
```

```
ASM int msg_eb29k_wait_for(void)
ASM int msg_eb030_wait_for(void)
```

ASM is used to denote that these labels are assembly-level labels, and have no leading underscore. The **msg_eb29k_wait_for** and **msg_eb030_wait_for** functions are called from the **msg_wait_for()** function in the message layer. For shared-memory interfaces, the target message drivers always receive messages in interrupt mode. Therefore, the **msg_eb29k_wait_for** and **msg_eb030_wait_for** functions return immediately with a return value of 0 (zero) to indicate no message in the receive buffer.

```
ASM void msg_intr(void)
```

ASM is used to denote that this label is an assembly-level label, and has no leading underscore. The interrupt handler for shared-memory interfaces is **msg_intr**, which is defined in the driver layer. The bootstrap code installs **msg_intr** as the interrupt handler for interrupts from the PC host to the target. Note that **montip** interrupts the target by writing to the “mailbox” register after copying the message to the target address space.

The interrupt handler, **msg_intr**, clears the interrupt by reading the “mailbox” register. The value read is then compared with FFh to determine whether **montip** interrupted to acknowledge receipt of the message from the target, or whether a new message was sent by **montip** to the target. If the content of the “mailbox” register is not FFh, then **msg_intr** posts a message interrupt to the message system by jumping to the **msg_V_arrive** label inside the message layer. **msg_V_arrive** is a virtual interrupt handler, which interrupts the debugger or the operating system based on the type of the message received.

The code below shows the **msg_intr** interrupt handler for the EB29030 board. The “mailbox” register is at offset 90000000h for the EB29030 board and at offset 80800000h for the EB29K board.

```
;
---------------------------------------------------MSG_INTR
       .equ mailbox,0x90000000
; interrupt vector for interrupts from PC host.
msg_intr:
       const  gr4, mailbox
       consth gr4, mailbox
       load   0, 0, gr4, gr4 ; clear interrupt,read mailbox
       and    gr4, gr4, 0xFF ; test for new message
       cpeq   gr4, gr4, 0xFF ; compare with 0xFF
       jmpf   gr4, msg_V_arrive  ; yes, interrupt msg
system
       nop
       ; no clear receive interrupt from montip.
       const  gr4, mailbox
       consth gr4, mailbox
       store  0, 0, gr4, gr4 ; clear interrupt
       iret
```

Target Serial-Interface Drivers

The drivers for the Z8530 serial communications controller and for AMD’s 29K Family microcontroller’s internal serial port are included with the MiniMON29K product software. The Z8530 SCC drivers are linked with the target monitor software for AMD’s EZ-030 board, and the Am29200 and Am29205 SCC drivers are linked with the target monitor software for the SA-29200 and SA-29205 boards.

The SA-29200 and SA-29205 message driver is explained in more detail on the following pages, and in Appendix C.

NOTE: Every message is appended with a 32-bit checksum value, which is the sum of all the bytes in the message (see page 5–2 for more information on checksums).

The SA-29200 and SA-29205 Message Driver

```
ASM int msg_initcomm(void)
```

ASM is used to denote that this label is an assembly-level label, and has no leading underscore. The **msg_initcomm** function for the SA-29200 or SA-29205 board is called from the message-layer initialization function, **msg_init()**. The interrupt handler for the interrupt line used by the Am29200 and Am29205 SCC, **serial_int**, must be installed during the bootstrap process. The **msg_initcomm** function installs the **msg_scc200_write** and **msg_scc200_wait_for** driver functions to write a message and wait for a message across the communications interface, respectively. As the interrupt line, INTR3, used by the serial port on the Am29200 or Am29205 microcontroller is shared by other internal peripherals, the interrupt handler, **serial_int**, uses a table of vectors, **intr3_V_table**. The handlers for the interrupts corresponding to the Am29200 or Am29205 serial port are installed into this table. **msg_initcomm** also installs a default handler to ignore the interrupts generated by unused peripherals.

To support parallel-port download from a PC to an SA-29200 or SA-29205 target mounted on an SA-29200 expansion board, the handler to receive a message through the Am29200 or Am29205 parallel port is also installed.

msg_initcomm calls the routines to initialize the serial port and the parallel port of the Am29200 or Am29205 microcontroller. It returns the version number of the communications drivers to the caller.

The code below shows the **msg_initcomm** routine for the SA-29200 and SA-29205 board.

```
; ---------------------------------------MSG_INITCOMM
; return version in gr96.
    .equ    TXDI_OFFSET, (31–5)*4
    .equ    RXDI_OFFSET, (31–6)*4
    .equ    RXSI_OFFSET, (31–7)*4
    .equ    PPI_OFFSET, (31–11)*4
    .externmsg_write_p
    .externmsg_wait_for_p
    .externmsg_scc200_init
    .externmsg_lpt200_init
msg_initcomm:
    const   gr96, save_regs
    consth  gr96, save_regs
    store   0, 0, gr97, gr96     ; backup gr97
    add     gr96, gr96, 4
    store   0, 0, gr98, gr96     ; backup gr98
    add     gr96, gr96, 4
    store   0, 0, lr0, gr96      ; backup lr0

    ;initialize the msg_write_p with write functions.
    const   gr96, msg_write_p
    consth  gr96, msg_write_p
    const   gr97, msg_scc200_write
    consth  gr97, msg_scc200_write
    store   0, 0, gr97, gr96     ; only one for now.

    ; initialize msg_wait_for_p pointer
    const   gr96, msg_wait_for_p
    consth  gr96, msg_wait_for_p
    const   gr97, msg_scc200_wait_for
    consth  gr97, msg_scc200_wait_for
    store   0, 0, gr97, gr96

    ; initialize table with default entries.
    const   gr96, intr3_V_table
    consth  gr96, intr3_V_table
    const   gr97, default_intr3
    consth  gr97, default_intr3
    const   gr98, 32–2
$1:
    store   0, 0, gr97, gr96
    jmpfdecgr98, $1
    add     gr96, gr96, 4
```

```
; install known handlers.
const  gr96, intr3_V_table+TXDI_OFFSET
consth gr96, intr3_V_table+TXDI_OFFSET
const  gr97, msg_scc200_tx_intr
consth gr97, msg_scc200_tx_intr
store  0, 0, gr97, gr96    ; tx intr

const  gr96, intr3_V_table+RXDI_OFFSET
consth gr96, intr3_V_table+RXDI_OFFSET
const  gr97, msg_scc200_rx_intr
consth gr97, msg_scc200_rx_intr
store  0, 0, gr97, gr96    ; rx intr

const  gr96, intr3_V_table+PPI_OFFSET
consth gr96, intr3_V_table+PPI_OFFSET
const  gr97, msg_ppi200_intr
consth gr97, msg_ppi200_intr
store  0, 0, gr97, gr96    ; ppi intr

; initialize the peripherals.
const  gr96, msg_scc200_init
consth gr96, msg_scc200_init
calli  lr0, gr96
nop

; initialize 29200 parallel port
const  gr96, msg_lpt200_init
consth gr96, msg_lpt200_init
calli  lr0, gr96
nop

; restore registers
const  gr96, save_regs
consth gr96, save_regs
load   0, 0, gr97, gr96    ; restore gr97
add    gr96, gr96, 4
load   0, 0, gr98, gr96    ; restore gr98
add    gr96, gr96, 4
load   0, 0, lr0, gr96     ; restore lr0

jmpi   lr0
const  gr96, COMM_VERSION  ; return version number
```

```
ASM void msg_scc200_write(msg_t *msg, int nbytes)
```

ASM is used to denote that this label is an assembly-level label, and has no leading underscore. The **msg_scc200_write** function is called from **msg_send()** to send the message contained in the **msg** buffer. The **nbytes** parameter gives the number of bytes to send. When **msg_scc200_write** is called, it checks the message in the **msg** buffer for an ACK or NACK message. If the message is not an ACK or NACK message, then **msg_scc200_write** computes the checksum of the message, and appends the checksum (32-bit value) to the end of the message. There is no checksum for ACK and NACK messages. The total number of bytes to write including the checksum bytes, if applicable, is stored in a static variable, **nbytes_to_write**. The value at **nbytes_to_write** is decremented by one after every byte is sent out to the host. Another static variable, **nextchar_p**, is used to point to the next character to be written out of the serial port. The **nextchar_p** variable is initialized with the starting address of the message to send.

The first byte of the message is then transmitted out of the serial port. If the drivers are built for interrupt driven mode, the **msg_scc200_write** function returns after transmitting the first byte. The remaining bytes are transmitted at the occurrence of the transmit interrupts. The **nbytes_to_write** and **nextchar_p** variables are updated by the transmit interrupt handlers.

If the drivers are built for polled mode, the **msg_scc200_write** function loops until all the message bytes are written out of the serial port.

The code below shows the **msg_scc200_write** function for the SA-29200 and SA-29205 board.

```
; -----------------------------------MSG_SCC200_WRITE
msg_scc200_write:
; In interrupt mode, it sends out the first character
; and returns. In this mode it is called with
; interrupts disabled. Interrupts are enabled after
; this call returns. In polled mode, it loops until
; the entire message is written.
; Called from msg_send. return via lr0.
; lr2 – pointer to message
; lr3 = nbytes in message.
    const   gr4, scc200_tmp_regs
    consth  gr4, scc200_tmp_regs
    store   0, 0, gr96, gr4      ; backup gr96
    const   gr96, scc200_tmp_regs+4
    consth  gr96, scc200_tmp_regs+4
    store   0, 0, gr97, gr96     ; backup gr97
    add     gr96, gr96, 4
    store   0, 0, gr98, gr96     ; backup gr98

    ; set nextchar_p
    const   gr96, nextchar_p
    consth  gr96, nextchar_p
    store   0, 0, lr2, gr96      ; next char to send.

    ; check the type of message,
    ; ack/nack have no checksum
    const   gr96, nbytes_to_write
    consth  gr96, nbytes_to_write
    load    0, 0, gr98, lr2
    jmpt    gr98, acknack_code
    add     gr97, lr3, 0
    add     gr97, lr3, 4         ; add checksum size
    store   0, 0, gr97, gr96     ; write nbytes to send

    ; compute checksum and append to end of message.
    add     gr96, lr2, 4
    load    0, 0, gr96, gr96     ; msg len
    add     gr96, gr96, 8        ; add msg size
    sub     gr96, gr96, 2
    const   gr98, 0              ; initialize checksum
$1:
    load    0, 1, gr97, lr2
    add     gr98, gr98, gr97
    jmpfdec gr96, $1
    add     lr2, lr2, 1
```

```
    ; append at lr2
    srl    gr97, gr98, 24
    store  0, 1, gr97, lr2
    srl    gr97, gr98, 16
    and    gr97, gr97, 0xff
    add    lr2, lr2, 1
    store  0, 1, gr97, lr2
    srl    gr97, gr98, 8
    and    gr97, gr97, 0xff
    add    lr2, lr2, 1
    store  0, 1, gr97, lr2
    and    gr97, gr98, 0xff
    add    lr2, lr2, 1
    store  0, 1, gr97, lr2

    ; Start sending out the message. This layer does
    ; not buffer the message. Instead it relies on
    ; the message remaining there until it is sent. A
    ; semaphore msg_send_p is cleared when the
    ; message is sent.

    ; wait for transmit holding register to empty.
    const  gr96, SPST
tx_loop:
    consth gr96, SPST
    load   0, 0, gr96, gr96     ; read status
    sll    gr96, gr96, (31 – THREShift)
    jmpf   gr96, tx_loop
    const  gr96, SPST

    ; get character from nextchar_p
    const  gr96, nextchar_p
    consth gr96, nextchar_p
    load   0, 0, gr97, gr96
    load   0, 1, gr98, gr97     ;get character to send
    add    gr97, gr97, 1        ; update
    store  0, 0, gr97, gr96     ; nextchar_p++

    ; stuff character
    const  gr96, SPTH
    consth gr96, SPTH
    store  0, 0, gr98, gr96     ; put char
```

```
    ; decrement nbytes_to_write
    const  gr96, nbytes_to_write
    consth gr96, nbytes_to_write
    load   0, 0, gr97, gr96
    sub    gr97, gr97, 1
    store  0, 0, gr97, gr96    ; nbytes_to_write––

   .ifdef SERIAL_POLL
    cpeq   gr98, gr97, 0       ; nbytes_to_write == 0?
    jmpt   gr98, $2            ; yes, then done.
    nop
    jmp    tx_loop
    const  gr96, SPST
   .endif

$2:
    const  gr96, firstmsg_flag
    consth gr96, firstmsg_flag
    load   0, 0, gr96, gr96
    jmpf   gr96, restore_regs
    const  gr96, _msg_sbuf_p
    consth gr96, _msg_sbuf_p
    const  gr97, 0
    store  0, 0, gr97, gr96
       ; clear _msg_sbuf_p for 1st msg.
    const  gr96, firstmsg_flag
    consth gr96, firstmsg_flag
    const  gr97, 0
    store  0, 0, gr97, gr96    ; clear firstmsg_flag
restore_regs:
   .ifdef SERIAL_POLL
    ; clear msg_sbuf_p
    const  gr96, _msg_sbuf_p
    consth gr96, _msg_sbuf_p
    const  gr97, 0
    store  0, 0, gr97, gr96    ; clear msg_sbuf_p
   .endif
    ; restore gr96–gr98
    const  gr96, scc200_tmp_regs+4
    consth gr96, scc200_tmp_regs+4
    load   0, 0, gr97, gr96    ; restore gr97
    add    gr96, gr96, 4
    load   0, 0, gr98, gr96    ; restore gr98
    const  gr96, scc200_tmp_regs
    consth gr96, scc200_tmp_regs
    load   0, 0, gr96, gr96    ; restore gr96
```

```
    jmpi   lr0
    nop

acknack_code:
    store  0, 0, gr97, gr96    ; write nbytes to send
    add    lr2, lr2, 4
    load   0, 0, gr96, lr2
    const  gr97, ack_flag
    consth gr97, ack_flag
    jmpt   gr96, set_nack_flag
    constn gr98, –1
    store  0, 0, gr98, gr97    ; set ack_flag

    jmp    tx_loop
    const  gr96, SPST

set_nack_flag:
    const  gr97, nack_flag
    consth gr97, nack_flag
    constn gr98, –1
    store  0, 0, gr98, gr97    ; set nack_flag

    jmp    tx_loop
    const  gr96, SPST
```

`ASM int msg_scc200_wait_for(void)`

ASM is used to denote that this label is an assembly-level label, and has no leading underscore. The **msg_scc200_wait_for** function returns immediately when the drivers are built for interrupt mode. It returns a value of 0 (zero) to indicate no message in the buffer. When the drivers are built for polled mode, the **msg_scc200_wait_for** function polls the serial-port status register of the Am29200 or Am29205 microcontroller for an incoming message byte. When a message byte is received, the received byte is stored in the receive buffer, **_msg_rbuf**. The **_msg_rbuf** receive buffer is then examined for a valid message. The functionalities of the **msg_scc200_wait_for** function is similar to the receive interrupt handler routine, **msg_scc200_rx_intr**, explained below.

`ASM void serial_int(void)`

ASM is used to denote that this label is an assembly-level label, and has no leading underscore. The interrupt handler to handle the Am29200 or Am29205 interrupts on the INTR3 line is **serial_int**. The bootstrap code must install **serial_int** as the interrupt handler for the INTR3 line.

The **serial_int** interrupt handler reads the Interrupt Control Register (ICT) of the Am29200 or Am29205 microcontroller to determine the cause of the interrupt. It then calls the appropriate handler routine from the **intr3_V_table**, which was initialized by the **msg_initcomm** function. The default interrupt handler, **default_intr3**, is called for interrupts generated by unused peripherals.

The code below shows how the **serial_int** interrupt handler is used.

```
; ---------------------------------------- SERIAL_INT
serial_int:
; We use count of leading zeroes to determine the
; offset in the interrupt table, and branch to the
; interrupt handler.
    const  gr4, intr_save
    consth gr4, intr_save
    store  0, 0, gr96, gr4      ; backup gr96

    const  gr96, intr_save+4
    consth gr96, intr_save+4
    store  0, 0, gr97, gr96     ; backup gr97
    const  gr96, ICT
    consth gr96, ICT
    load   0, 0, gr96, gr96     ; read ICT
    clz    gr96, gr96
    cpeq   gr97, gr96, 32
    jmpt   gr97, $2             ; no interrupts??
    nop
    sll    gr96, gr96, 2      ; find offset into table
    const  gr97, intr3_V_table
    consth gr97, intr3_V_table
    add    gr97, gr97, gr96  ; handler address pointer

    const  gr96, intr_save
    consth gr96, intr_save
    load   0, 0, gr96, gr96     ; restore gr96

    load   0, 0, gr4, gr97      ; address

    const  gr97, intr_save+4
    consth gr97, intr_save+4
    load   0, 0, gr97, gr97     ; restore gr97
```

```
    jmpi   gr4
    nop
$2:
    ; restore regs
    const  gr96, intr_save+4
    consth gr96, intr_save+4
    load   0, 0, gr97, gr96    ; restore gr97
    const  gr96, intr_save
    consth gr96, intr_save
    load   0, 0, gr96, gr96    ; restore gr96
    iret
```

The transmit interrupt handler, **msg_scc200_tx_intr**, examines the **nbytes_to_write** variable to determine if there are any more bytes to write. If there are more bytes to write, then **msg_scc200_tx_intr**:

1. Decrements **nbytes_to_write** by one.
2. Gets the byte pointed to by **nextchar_p**.
3. Increments **nextchar_p** by one to point to the next byte.
4. Sends the next character out of the serial port.
5. Returns from the interrupt handler.

If no more bytes remain to be written, then the transmit interrupt routine checks the **ack_flag** to determine if the message just written out was an ACK message. If an ACK message was written out, it posts an interrupt to the message system by jumping to the label **msg_V_arrive** inside the message layer. For other messages, it simply returns from the interrupt handler.

The code below shows the transmit interrupt handler.

```
msg_scc200_tx_intr:
    const  gr4, intr_tmp_regs
    consth gr4, intr_tmp_regs
    store  0, 0, gr96, gr4     ; backup gr96
    const  gr96, intr_tmp_regs+4
    consth gr96, intr_tmp_regs+4
    store  0, 0, gr97, gr96    ; backup gr97
    add    gr96, gr96, 4
    store  0, 0, gr98, gr96    ; backup gr98
```

```
    const   gr96, ICT
    consth  gr96, ICT
    const   gr97, TXDI
    consth  gr97, TXDI
    store   0, 0, gr97, gr96     ; clear TXDI

    ; check for more bytes to send.
    const   gr96, nbytes_to_write
    consth  gr96, nbytes_to_write
    load    0, 0, gr97, gr96 ; get bytes left
    cpeq    gr98, gr97, 0        ; compare with zero
    jmpt    gr98, $3             ; yes, none left check
                                 ; nack/ack
    nop

    ; get next byte
    sub     gr97, gr97, 1
    store   0, 0, gr97, gr96     ; nbytes_to_write--
    const   gr96, nextchar_p
    consth  gr96, nextchar_p
    load    0, 0, gr97, gr96
    load    0, 1, gr98, gr97     ; get character
    add     gr97, gr97, 1
    store   0, 0, gr97, gr96     ; nextchar_p++

    ; stuff byte
    const   gr96, SPTH
    consth  gr96, SPTH
    store   0, 0, gr98, gr96     ; put char
$4:
    ; restore gr96–gr98 registers.
    const   gr96, intr_tmp_regs+4
    consth  gr96, intr_tmp_regs+4
    load    0, 0, gr97, gr96     ; restore gr97
    add     gr96, gr96, 4
    load    0, 0, gr98, gr96     ; restore gr98
    const   gr96, intr_tmp_regs
    consth  gr96, intr_tmp_regs
    load    0, 0, gr96, gr96     ; restore gr96

    iret
```

```
$3:
    ; check ack_flag if one just sent and clear it.
    const  gr96, ack_flag
    consth gr96, ack_flag
    load   0, 0, gr97, gr96    ; get flag
    jmpt   gr97, valid_msg     ; set, valid msg intr
    nop
    jmp    $4
    nop

valid_msg:
    ; clear ack_flag
    const  gr97, 0
    store  0, 0, gr97, gr96    ; clear flag

    ; restore gr96–gr98 registers.
    const  gr96, intr_tmp_regs+4
    consth gr96, intr_tmp_regs+4
    load   0, 0, gr97, gr96    ; restore gr97
    add    gr96, gr96, 4
    load   0, 0, gr98, gr96    ; restore gr98
    const  gr96, intr_tmp_regs
    consth gr96, intr_tmp_regs
    load   0, 0, gr96, gr96    ; restore gr96
    jmp    msg_V_arrive        ; post interrupt to
    nop                        ; message system
```

The receive interrupt handler, **msg_scc200_rx_intr**, reads the character from the serial port and stores it where the **_msg_next_p** variable is pointing to in the receive buffer, **_msg_rbuf**. The **_msg_next_p** pointer is then incremented by one. The receive buffer is then examined to determine if a valid message has been received. A valid message can be either a MiniMON29K message, or the ACK or NACK message. If the receive buffer does not have the complete message, the receive interrupt handler returns from the interrupt handler, and waits for more incoming bytes.

If an ACK message is received, then the receive interrupt routine resets the **_msg_sbuf_p** semaphore to zero, freeing up the message channel for subsequent messages to be sent.

If a NACK message is received, then the receive interrupt routine calls the **msg_scc200_write** routine with a pointer to the message last sent, which is stored in a static variable, **_msg_lastsent_p**.

If a valid MiniMON29K message is received, the receive interrupt routine computes the checksum of the message bytes received. It then compares the checksum computed with the checksum value received from the host. If the checksums compare to be the same, then it calls the **msg_scc200_write** function to send an ACK message to the host. If the checksums are not equal, then it calls the **msg_scc200_write** function to send a NACK message to the host.

The code below shows the receive interrupt handler, **msg_scc200_rx_intr**.

```
; ----------------------------------------MSG_SCC200_RX_INTR
msg_scc200_rx_intr:
    const  gr4, intr_tmp_regs
    consth gr4, intr_tmp_regs
    store  0, 0, gr96, gr4      ; backup gr96
    const  gr96, intr_tmp_regs+4
    consth gr96, intr_tmp_regs+4
    store  0, 0, gr97, gr96     ; backup gr97
    add    gr96, gr96, 4
    store  0, 0, gr98, gr96     ; backup gr98

    const  gr96, ICT
    consth gr96, ICT
    const  gr97, RXDI
    consth gr97, RXDI
    store  0, 0, gr97, gr96     ; clear RXDI

    ; receive the character and put in buffer.
    const  gr96, SPRB
    consth gr96, SPRB
    load   0, 0, gr96, gr96     ; gr96 has received character

handle_rx_char:
    ; put in _msg_next_p location.
    const  gr97, _msg_next_p
    consth gr97, _msg_next_p
    load   0, 0, gr98, gr97
    store  0, 1, gr96, gr98     ; save character
    add    gr98, gr98, 1        ; update _msg_next_p
    store  0, 0, gr98, gr97

    ; check the buffer for a minimon message.
    const  gr96, _msg_next_p
    consth gr96, _msg_next_p
    load   0, 0, gr97, gr96     ; msg_next_p
    const  gr96, _msg_rbuf
    consth gr96, _msg_rbuf      ; msg_rbuf
    sub    gr98, gr97, gr96     ; msg_rbuf–msg_next_p = len
```

```
    cplt   gr97, gr98, 8       ; len < 8
    jmpf   gr97,check_for_msg  ; no, check for message.
    nop

do_iret:
    ; restore gr96–gr98 registers
    const  gr96, intr_tmp_regs+4
    consth gr96, intr_tmp_regs+4
    load   0, 0, gr97, gr96    ; restore gr97
    add    gr96, gr96, 4
    load   0, 0, gr98, gr96    ; restore gr98
    const  gr96, intr_tmp_regs
    consth gr96, intr_tmp_regs
    load   0, 0, gr96, gr96    ; restore gr96

    iret

check_for_msg:
    ; a message header is in buffer.
    ; gr98 has total length.
    ; gr96 has msg_rbuf
    load   0, 0, gr97, gr96    ; get msg code
    jmpt   gr97, ack_nack_recd  ; handle ack/nack msg.
    nop

; ----------------------------------------------------------
; message.
    const  gr96, _msg_rbuf+4
    consth gr96, _msg_rbuf+4
    load   0, 0, gr96, gr96    ; msg length
    add    gr96, gr96, 8+4
           ; add msg header size and checksum
    cpgeu  gr97, gr98, gr96
           ; have we received all the bytes.
    jmpf   gr97, do_iret       ; no return
    nop

    ; compute checksum for message
    const  gr97, intr_tmp_regs+3*4
    consth gr97, intr_tmp_regs+3*4
    store  0, 0, gr99, gr97    ; backup gr99
    const  gr99, 0             ; initialize checksum
    sub    gr96, gr96, 4       ; sub checksum size
    const  gr97, _msg_rbuf
    consth gr97, _msg_rbuf
```

```
    sub    gr96, gr96, 2
$6:
    load   0, 1, gr98, gr97
    add    gr99, gr99, gr98
    jmpfdec    gr96, $6
    add    gr97, gr97, 1

    ; get checksum send by montip
    load   0, 1, gr96, gr97
    sll    gr96, gr96, 24
    add    gr97, gr97, 1
    load   0, 1, gr98, gr97
    sll    gr98, gr98, 16
    or     gr96, gr96, gr98
    add    gr97, gr97, 1
    load   0, 1, gr98, gr97
    sll    gr98, gr98, 8
    or     gr96, gr96, gr98
    add    gr97, gr97, 1
    load   0, 1, gr98, gr97
    or     gr96, gr96, gr98

    cpeq   gr97, gr96, gr99    ; compare checksums
    ; reset msg_next_p to beginning of msg_rbuf
    const  gr96, _msg_rbuf
    consth gr96, _msg_rbuf
    const  gr98, _msg_next_p
    consth gr98, _msg_next_p
    jmpt   gr97, ack_it        ; same, valid message
    store  0, 0, gr96, gr98    ; reset msg_next_p

    ; send a nack msg to montip.
    ; restore gr96–gr99 registers
    const  gr96, intr_tmp_regs+4
    consth gr96, intr_tmp_regs+4
    load   0, 0, gr97, gr96    ; restore gr97
    add    gr96, gr96, 4
    load   0, 0, gr98, gr96    ; restore gr98
    add    gr96, gr96, 4
    load   0, 0, gr99, gr96    ; restore gr99
    const  gr96, intr_tmp_regs
    consth gr96, intr_tmp_regs
    load   0, 0, gr96, gr96    ; restore gr96
```

```
        ; save lr0, lr2, lr3
        const  gr4, intr_tmp_regs
        consth gr4, intr_tmp_regs
        store  0, 0, lr0, gr4      ; save lr0
        const  lr0, intr_tmp_regs+4
        consth lr0, intr_tmp_regs+4
        store  0, 0, lr2, lr0      ; save lr2
        add    lr0, lr0, 4
        store  0, 0, lr3, lr0      ; save lr3

        const  lr2, nack_msg_p
        consth lr2, nack_msg_p
        const  lr3, 8
        call   lr0, msg_scc200_write
        nop

        const  lr0, intr_tmp_regs+4
        consth lr0, intr_tmp_regs+4
        load   0, 0, lr2, lr0      ; restore lr2
        add    lr0, lr0, 4
        load   0, 0, lr3, lr0      ; restore lr3
        const  lr0, intr_tmp_regs
        consth lr0, intr_tmp_regs
        load   0, 0, lr0, lr0      ; restore lr0

        iret

    ack_it:
        ; restore gr96–gr99 registers
        const  gr96, intr_tmp_regs+4
        consth gr96, intr_tmp_regs+4
        load   0, 0, gr97, gr96    ; restore gr97
        add    gr96, gr96, 4
        load   0, 0, gr98, gr96    ; restore gr98
        add    gr96, gr96, 4
        load   0, 0, gr99, gr96    ; restore gr99
        const  gr96, intr_tmp_regs
        consth gr96, intr_tmp_regs
        load   0, 0, gr96, gr96    ; restore gr96

        ; send an ack to montip
        ; save lr0, lr2, lr3
        const  gr4, intr_tmp_regs
        consth gr4, intr_tmp_regs
        store  0, 0, lr0, gr4      ; save lr0
        const  lr0, intr_tmp_regs+4
        consth lr0, intr_tmp_regs+4
        store  0, 0, lr2, lr0      ; save lr2
        add    lr0, lr0, 4
        store  0, 0, lr3, lr0      ; save lr3
```

```
    const  lr2, ack_flag
    consth lr2, ack_flag
    constn lr3, -1
    store  0, 0, lr3, lr2      ; set ack_flag

    const  lr2, ack_msg_p      ; pointer to ack msg str
    consth lr2, ack_msg_p
    const  lr3, 8              ; nbytes in ack msg.
    call   lr0, msg_scc200_write ; sends the first character
    nop                        ; and returns.

    const  lr0, intr_tmp_regs+4
    consth lr0, intr_tmp_regs+4
    load   0, 0, lr2, lr0      ; restore lr2
    add    lr0, lr0, 4
    load   0, 0, lr3, lr0      ; restore lr3
    const  lr0, intr_tmp_regs
    consth lr0, intr_tmp_regs
    load   0, 0, lr0, lr0      ; restore lr0
    iret
; ------------------------------------------------------------
ack_nack_recd:
    const  gr96, _msg_rbuf
    consth gr96, _msg_rbuf
    const  gr97, _msg_next_p
    consth gr97, _msg_next_p
    store  0, 0, gr96, gr97    ; initialize msg_next_p

    add    gr96, gr96, 4
    load   0, 0, gr97, gr96    ; get msg len field
    jmpf   gr97, ack_recd      ; ack received.
    nop

nack_recd:
    ; restore gr96-gr99 registers
    const  gr96, intr_tmp_regs+4
    consth gr96, intr_tmp_regs+4
    load   0, 0, gr97, gr96    ; restore gr97
    add    gr96, gr96, 4
    load   0, 0, gr98, gr96    ; restore gr98
    const  gr96, intr_tmp_regs
    consth gr96, intr_tmp_regs
    load   0, 0, gr96, gr96    ; restore gr96
```

```
    ; save lr0, lr2, lr3
    const  gr4, intr_tmp_regs
    consth gr4, intr_tmp_regs
    store  0, 0, lr0, gr4      ; save lr0
    const  lr0, intr_tmp_regs+4
    consth lr0, intr_tmp_regs+4
    store  0, 0, lr2, lr0      ; save lr2
    add    lr0, lr0, 4
    store  0, 0, lr3, lr0      ; save lr3

    const  lr2, _msg_lastsent_p  ; address of msg
    consth lr2, _msg_lastsent_p
    load   0, 0, lr2, lr2
    add    lr3, lr2, 4
    load   0, 0, lr3, lr3      ; msg length
    add    lr3, lr3, 8         ; msglen+msg header
    call   lr0, msg_scc200_write
    nop

    const  lr0, intr_tmp_regs+4
    consth lr0, intr_tmp_regs+4
    load   0, 0, lr2, lr0      ; restore lr2
    add    lr0, lr0, 4
    load   0, 0, lr3, lr0      ; restore lr3
    const  lr0, intr_tmp_regs
    consth lr0, intr_tmp_regs
    load   0, 0, lr0, lr0      ; restore lr0

    iret

ack_recd:
    ; clear _msg_sbuf_p semaphore
    const  gr96, _msg_sbuf_p
    consth gr96, _msg_sbuf_p
    const  gr97, 0
    store  0, 0, gr97, gr96    ; clear semaphore

    jmp    do_iret
    nop
```

Chapter 5

MiniMON29K Messages

This chapter first describes an extension to the message system, which ensures reliable serial communications at higher baud rates. The chapter then describes the structure of the standard MiniMON29K messages, and lists each message, grouped by type. The chapter sections are as follows:

NOTE: Throughout this chapter, “target” refers to the 29K Family-based target system running the MiniMON29K monitor software; “host” refers to the computer system running **montip**.

Message Checksum Tags for Serial Communications

The MiniMON29K product extends the message communication protocol, when doing serial communications, with checksums that are used to ensure reliable serial communications at higher baud rates. This extra layer of protocol is only used for the serial-communication drivers and is not used for the shared-memory message-system drivers.

Protocol

Every message from either the host or target is appended with a 32-bit checksum word. The checksum is the sum of every byte in the message including the header.

The serial-communications drivers append a 32-bit checksum at the end of the message. The checksum is the sum of all the bytes of the message. The receiver of the message computes a checksum of the received message bytes and compares it with the checksum value received. If the checksums are equal, then the receiver sends a Checksum ACK message to acknowledge the receipt of a valid MiniMON29K message. If the checksums are not equal, then the receiver sends a Checksum NACK message, indicating a transmission error in the received message. The Checksum ACK/Checksum NACK messages are used by the communications drivers to report transmission errors and to resend messages.

Checksum ACK Message

```
0xffffffff
0x00000000
```

Checksum NACK Message

```
0xffffffff
0xffffffff
```

Implementation

The Checksum ACK/Checksum NACK messages are independent of the MiniMON29K messages, and are handled by the communications drivers. However, the message buffers are large enough to provide enough space at the end of the message to hold the 32-bit message checksum.

Example

The following example shows the messages sent and received when **montip** establishes a synchronous connection. The sequence is as follows:

1. Target sends a HALT message (when powered up).
2. Host sends a Checksum ACK.
3. Host sends a CONFIG_REQ message.
4. Target sends a Checksum ACK.
5. Target responds with a CONFIG message.
6. Host sends a Checksum ACK.
7. A synchronous connection is established.

```
Target: halt message (on power up)
0000002b
0000000f
00000007
00001834
00001830
00000000
000000d6 checksum

Host: checksum ack
FFFFFFFF
00000000

Host:
00000001 config request
00000000
00000001 checksum

Target: checksum ack
FFFFFFFF
00000000
```

```
Target: config response
00000021
00000030
00000003
05040512
00000000
00080000
00000000
00080000
00000000
00080000
00000200
0000001e
00000000
00000002
000000ae checksum

Host: checksum ack
FFFFFFFF
00000000
```

MiniMON29K Message Description

The message structure, byte ordering, definition, classification, passing protocol, and numbers are described in this section.

Message Structure

The basic message takes the following form:

```
struct <message_name> {
         INT32  code;
         INT32  length;
         <parameter 1>;
         <parameter 2>;
                  .
                  .
                  .
         <parameter n>;
         };
```

The first field in the message is *code*. This is a 32-bit integer. Each type of message in the system is given a unique identification code. This allows the receiver of the message to determine what sort of message is arriving even before the entire message is read.

The second field is the *length* of the parameter list. This is also a 32-bit integer, and is measured in bytes. The *length* is not the length of the entire message; the *code* and *length* fields are not included. For example, a message containing no parameters has a *length* of 0. The entire message, however, will have a length of 8 bytes because the *code* and *length* fields are always a part of the message.

This format provides a convenient method of transferring messages between the target and the host.

Some systems may have restrictions on the amount of message buffer space available. For this reason, a maximum message length is specified by the target. It is the responsibility of the host to keep the size of the messages smaller than this maximum message size. The target should, however, detect messages of illegal lengths, both incoming and outgoing, and respond with the proper error message.

Byte Ordering

The MiniMON29K messages are defined as a stream of bytes. All MiniMON29K messages are transmitted in the same byte order, or endian type, as that of the 29K Family-based target. (See the **–le** option on page 1–3 if your 29K Family-based target is little endian.)

All message fields are 32 bits. The only exceptions are the arrays of bytes (data) at the end of the messages, which require the transfer of data. This format makes endian conversion simple.

Message Definition

The following sections contain the definition of each of the messages, including the message structure, parameters, and possible error conditions.

The structure of the messages and all examples of code that follow will be in C. Also, because the physical structure of the message is important, some basic data types have been used to describe the messages. These types are:

- **INT32** — This is a 32-bit integer.
- **ADDR32** — This is a 32-bit address. This is physically represented the same as **INT32**, but it is unsigned.
- **BYTE** — This is an 8-bit quantity, usually equivalent to **unsigned char**.
- **BOOLEAN** — This is also a 32-bit integer. FALSE is defined as 0 and TRUE is defined as 1. A 32-bit quantity is used to maintain 32-bit word alignment.

Message Classification

Messages 0 through 127 are reserved for AMD's use. These messages are divided in the following manner:

- Messages 0 through 63 are classified as debug messages, and are described beginning on page 5–15. These messages are transmitted between the host and the MiniMON29K monitor on the target. Of these, some are sent from the host to the target and some are sent from the target to the host.
- Messages 64 through 127 are classified as operating-system messages, and are described beginning on page 5–52. These messages are transmitted between the host and the application/operating system running on the target. In the default configuration of the MiniMON29K monitor, the HIF kernel of **osboot** transmits and receives the operating-system messages on the target.

Any message number greater than 127 may be used for custom messages.

Message-Passing Protocol

The communication between host and target takes place by passing synchronous message pairs. Typically, the host sends a request message to the target, and the target sends an acknowledgement back to the host. This acknowledgement message may contain requested data, or the message may be a simple handshake acknowledgement. If the requested action cannot be successfully completed, an error message is returned as the acknowledgement.

The general pairing of messages is described in Table 5–4 on page 5–13 and in the sections on the individual messages that follow.

NOTE: The messages do not contain any checksum or error detection information. It is the responsibility of the communications driver to provide reliable, sequenced delivery of messages.

An example of message interaction between the target and the host is shown below. When the target system is powered up, this first message is sent:

```
0000002b   Message 0x2b = 43 halt message
00000010   0x10 = 16 bytes follow (4 words)
00000007   Memory space I
xxxxxxxx   pc0 value
xxxxxxxx   pc1 value
00000000   trap number
```

The target then loops, waiting for messages from the host. When **montip** is invoked by a debugger front end on the host system, it sends a configuration request message:

```
00000001   Message 0x1 = 1 config request
00000000   0x0 no bytes follow
```

montip then waits for an acknowledgement message from the target. When the target receives the configuration request message, it responds with the configuration message:

```
00000021   0x21 = 33 config message
00000030   0x30 = 48 bytes of information follow (12 words)
xxxxxxxx   Processor ID
00000010   Version number of debugger core
00000000   Starting address of instruction memory
0007ffff   Ending address of instruction memory
00000000   Starting address of data memory
0007ffff   Ending address of data memory
00000000   Starting address of ROM memory
0007ffff   Ending address of ROM memory
00000100   0x100 = 256 max size of target message buffer
0000000a   0xa = 10 breakpoints can be used
ffffffff   Coprocessor PRL. It is –1 if not present
00000002   Target OS version
```

When **montip** receives the configuration message, it also has synchronized with the target. In this way, **montip** and the target-resident monitor communicate by exchanging messages.

Message Numbers

The messages and their corresponding numeric codes are listed in the following tables. In these tables, “host” refers to the host computer running **montip** and “target” refers to the 29K Family-based hardware platform running the MiniMON29K monitor software. Table 5–1 lists all the messages in alphabetical order, with their corresponding decimal and hexadecimal number, and the page number on which the message can be found. Table 5–2 lists the host-to-target messages, with their corresponding numeric codes in both hexadecimal and decimal notation. Table 5–3 lists the target-to-host messages, with their corresponding numeric codes in both hexadecimal and decimal notation. Table 5–4 lists the requestor messages in alphabetical order, with each message’s corresponding acknowledgement message. The codes for the processor memory spaces used in the messages are listed in Table 5–5.

Table 5–1. Alphabetical List of Messages

Message	Decimal Number	Hexadecimal Number	Page Number
BKPT_RM	6	6	5–25
BKPT_RM_ACK	38	26	5–45
BKPT_SET	5	5	5–23
BKPT_SET_ACK	37	25	5–44
BKPT_STAT	7	7	5–26
BKPT_STAT_ACK	39	27	5–46
BREAK	13	D	5–35
CHANNEL0	65	41	5–54
CHANNEL0_ACK	97	61	5–60
CHANNEL1	98	62	5–61
CHANNEL1_ACK	66	42	5–55
CHANNEL2	99	63	5–62
CHANNEL2_ACK	67	43	5–56
CONFIG	33	21	5–36
CONFIG_REQ	1	1	5–17
COPY	8	8	5–27

Message	Decimal Number	Hexadecimal Number	Page Number
COPY_ACK	40	28	5–47
ERROR	63	3F	5–51
FILL	9	9	5–29
FILL_ACK	41	29	5–48
GO	11	B	5–33
HALT	43	2B	5–50
HIF_CALL	96	60	5–59
HIF_CALL_RTN	64	40	5–53
INIT	10	A	5–31
INIT_ACK	42	2A	5–49
READ_ACK	35	23	5–41
READ_REQ	3	3	5–19
RESET	0	0	5–16
STATUS	34	22	5–38
STATUS_REQ	2	2	5–18
STDIN_NEEDED	100	64	5–63
STDIN_NEEDED_ACK	68	44	5–57
STDIN_MODE_ACK	69	45	5–58
STDIN_MODE	101	65	5–64
STEP	12	C	5–34
WRITE_ACK	36	24	5–43
WRITE_REQ	4	4	5–21

Table 5–2. Host-to-Target Message Definitions

Hexadecimal Number	Decimal Number	Message
0	0	RESET
1	1	CONFIG_REQ
2	2	STATUS_REQ
3	3	READ_REQ
4	4	WRITE_REQ
5	5	BKPT_SET
6	6	BKPT_RM
7	7	BKPT_STAT
8	8	COPY
9	9	FILL
A	10	INIT
B	11	GO
C	12	STEP
D	13	BREAK
40	64	HIF_CALL_RTN
41	65	CHANNEL0
42	66	CHANNEL1_ACK
43	67	CHANNEL2_ACK
44	68	STDIN_NEEDED_ACK
45	69	STDIN_MODE_ACK

Table 5–3. Target-to-Host Message Definitions

Hexadecimal Number	Decimal Number	Message
21	33	CONFIG
22	34	STATUS
23	35	READ_ACK
24	36	WRITE_ACK
25	37	BKPT_SET_ACK
26	38	BKPT_RM_ACK
27	39	BKPT_STAT_ACK
28	40	COPY_ACK
29	41	FILL_ACK
2A	42	INIT_ACK
2B	43	HALT
3F	63	ERROR
60	96	HIF_CALL
61	97	CHANNEL0_ACK
62	98	CHANNEL1
63	99	CHANNEL2
64	100	STDIN_NEEDED
65	101	STDIN_MODE

Table 5–4. Requestor/Acknowledgement Message Correspondence

Requestor	Acknowledgement
BKPT_RM	BKPT_RM_ACK
BKPT_SET	BKPT_SET_ACK
BKPT_STAT	BKPT_STAT_ACK
BREAK	HALT
CHANNEL0	CHANNEL0_ACK
CHANNEL1	CHANNEL1_ACK
CHANNEL2	CHANNEL2_ACK
CONFIG_REQ	CONFIG
COPY	COPY_ACK
FILL	FILL_ACK
GO	HALT, or any target-to-host operating-system message
HIF_CALL	HIF_CALL_RTN
INIT	INIT_ACK
READ_REQ	READ_ACK
RESET	HALT
STATUS_REQ	STATUS
STDIN_NEEDED	STDIN_NEEDED_ACK
STDIN_MODE	STDIN_MODE_ACK
STEP	HALT
WRITE_REQ	WRITE_ACK
Any host-to-target message	ERROR

Table 5–5. Memory Spaces

Decimal Number	Hexadecimal Number	Memory Space
0	0	LOCAL_REG: Local processor register
1	1	GLOBAL_REG: Global processor register
2	2	SPECIAL_REG: Special processor register
3	3	TLB_REG: Translation lookaside buffer
4	4	COPROC_REG: Coprocessor register
5	5	I_MEM: Instruction memory
6	6	D_MEM: Data memory
7	7	I_ROM: Instruction ROM
8	8	D_ROM: Data ROM
9	9	I_O: Input/output
10	Ah	I_CACHE: Instruction cache
11	Bh	D_CACHE: Data cache
12	Ch	PC_SPACE: PC0, PC1
13	Dh	A_SPCL_REG: User special processor register
14	Eh	ABS_REG: Absolute register number
15	Fh	PC_RELATIVE: PC relative offsets
254	FEh	generic space

MiniMON29K Debug Messages

A set of messages is defined in the following sections. These messages provide the capability to control, probe, and modify the state of the system. With this capability, a variety of useful host functions may be implemented.

In addition to the basic functions, some useful but nonessential primitives are included. These primitives are included primarily as a convenience for the developers of host code.

It should also be mentioned that the message interface to the target provides the ability to add new functionality. This provides a natural path for extensions that will maintain upward compatibility.

Messages 0 through 63 are classified as debug messages. These messages are transmitted between the host and the MiniMON29K monitor on the target.

- Messages 0 through 31 are sent from the host to the target.
- Messages 32 through 63 are sent from the target to the host, and typically are acknowledgements.

The debug messages are listed on the following pages, in numerical order. See page 5–52 for the operating-system messages.

Message 0 (0h): RESET (Reset Processor)

Message

```
#define RESET 0

struct reset_msg_t {
         INT32    code; /* 0 */
         INT32    length;
          };
```

Direction

Host-to-target

Acknowledgement

HALT (on page 5–50)

Description

This message is used to reset the target processor. This is equivalent to resetting the hardware manually. This message has no parameters and will always have a *length* field of 0.

Message 1 (1h): CONFIG_REQ (Configuration Request)

Message

```
#define CONFIG_REQ 1

struct config_req_msg_t {
         INT32    code; /* 1 */
         INT32    length;
         };
```

Direction

Host-to-target

Acknowledgement

CONFIG (on page 5–36)

Description

This message is used to request configuration information from the target. This message has no parameters and will always have a *length* field of 0. The target should always respond to the CONFIG_REQ message with a CONFIG message.

For more on the information returned by CONFIG_REQ, see the description of the CONFIG message.

Message 2 (2h): STATUS_REQ (Status Request)

Message

```
#define STATUS_REQ 2

struct status_req_msg_t {
         INT32    code; /* 2 */
         INT32    length;
         };
```

Direction

Host-to-target

Acknowledgement

STATUS (on page 5–38)

Description

This message is used to get status information from the target. This message has no parameters and will always have a *length* field of 0. The target should always respond to the STATUS_REQ message with a STATUS message.

The STATUS_REQ message should be distinguished from the CONFIG_REQ message. The CONFIG_REQ message requests static configuration information, usually concerning the hardware. The STATUS_REQ message requests run-time statistics.

Some targets may not gather some or all of the data requested by STATUS_REQ. For more details on the information returned by the STATUS_REQ message, see the description of the STATUS message.

Message 3 (3h):
READ_REQ (Read Request)

Message

```
#define READ_REQ 3

struct read_req_msg_t {
         INT32    code; /* 3 */
         INT32    length;
         INT32    memory_space;
         ADDR32   address;
         INT32    count;
         INT32    size;
         };
```

where:

memory_space — Defines the memory space to be read. The codes used to specify the processor memory spaces are listed in Table 5–5 on page 5–14.

address — Is the address of the requested data in the data space. This address is a 32-bit quantity.

count — Is the number of objects to read.

size — Is the size of the object to read in bytes (1 = byte, 2 = half word, and 4 = full word).

Direction

Host-to-target

Acknowledgement

READ_ACK (on page 5–41)

Description

This message requests that some part of the state of the target be read. The host should never request more data than will fit in the message buffer. Larger requests should be broken up into several READ_REQ messages. If the host requests more data than will fit in a message buffer, an ERROR message is returned. It is also possible that part or all of the requested memory space is not accessible to the target processor. In this case, an ERROR message is returned to the host.

Message 4 (4h): WRITE_REQ (Write Request)

Message

```
#define WRITE_REQ 4

struct write_req_msg_t {
        INT32    code; /* 4 */
        INT32    length;
        INT32    memory_space;
        ADDR32   address;
        INT32    count;
        INT32    size
        BYTE     data[<byte_count>];
        };
```

where:

memory_space	Defines which memory space will be modified. The codes used to specify the processor memory spaces are listed in Table 5–5 on page 5–14.
address	Is the address in this memory space where the data is to be written.
count	Is the size of the data array in object sizes. This information is somewhat redundant to the message size parameter in the message header. It is included for convenience and for consistency with the READ_REQ message.
size	The size of the object in the data array (1 = byte, 2 = half word, and 4 = full word). Count*size = total length of the array data in bytes.
data	Is an array of bytes. These bytes will be written into the appropriate memory space starting at the specified address.

Direction

Host-to-target

Acknowledgement

WRITE_ACK (on page 5–43)

Description

This message requests that the state of the target be modified. When the data sent by the WRITE_REQ message is successfully written on the target, a WRITE_ACK message is returned in acknowledgement.

It is possible that part or all of the requested memory space is not accessible to the target processor. In this case, an ERROR message is returned to the host. If an error condition is encountered, the host can make no assumptions about the partial success of the request. The state of the processor may or may not have been modified.

It is also possible that the data sent by the WRITE_REQ will overflow the message buffer on the target. The host should be aware of the buffer size limitations of the target, and should not send such messages. Should too large a message be sent, however, it is the responsibility of the target to safely remove this message from the message stream and respond with an ERROR message.

Message 5 (5h): BKPT_SET (Set Breakpoint)

Message

```
#define BKPT_SET 5

struct bkpt_set_msg_t {
         INT32    code; /* 5 */
         INT32    length;
         INT32    memory_space;
         ADDR32   bkpt_addr;
         INT32    pass_count;
         INT32    bkpt_type;
         };
```

where:

memory_space Is the address space where the breakpoint is to be set. In most cases, this will be the instruction memory of the system.

bkpt_addr Is the address of the breakpoint.

pass_count Specifies the number of times the breakpoint must be encountered before control is passed from the application to the monitor. A *pass_count* of 1 means that a break should occur the next time the instruction at this address is executed.

bkpt_type Specifies the type of breakpoint to set:

–1 Specifies the software breakpoint, for example, replacing the instruction at the breakpoint location.

0 and 1 Specify the hardware breakpoint, for example, using the Am29050 breakpoint control registers. When 0 is specified as *bkpt_type*, the breakpoint comparison is performed when instruction translation is disabled. When 1 is specified as the *bkpt_type*, the breakpoint comparison is performed when instruction translation is enabled.

Direction

Host-to-target

Acknowledgement

BKPT_SET_ACK (on page 5–44)

Description

This message is sent by the host to set a breakpoint in the code. While it is possible to implement breakpoints using other primitives, BKPT_SET is included for convenience.

This message passes three parameters to the target. If the address specified by the *memory_space* and *address* parameters is not a valid writable address, an ERROR message will be returned.

The software predefines a limit of 48 on the number of breakpoints that can be set on the target. If an attempt is made to set a new breakpoint when this limit has been reached, an ERROR message will be returned to the host. When the breakpoint is successfully set on the target, a BKPT_SET_ACK message is returned by the target.

All positive *pass_counts* are interpreted as "sticky" breakpoints. A *pass_count* of 0 is interpreted as a "nonsticky" breakpoint. All negative numbers signify nonsticky breakpoints with a *pass_count* of the absolute value of the *pass_count* parameter.

Message 6 (6h): BKPT_RM (Remove Breakpoint)

Message

```
#define BKPT_RM 6

struct bkpt_rm_msg_t {
         INT32    code; /* 6 */
         INT32    length;
         INT32    memory_space;
         ADDR32   bkpt_addr;
         };
```

where:

memory_space	Is the address space where the breakpoint is to be removed.
bkpt_addr	Is the address of the breakpoint.

Direction

Host-to-target

Acknowledgement

BKPT_RM_ACK (on page 5–45)

Description

This message is used to remove a breakpoint from the system. The memory space and address of the breakpoint are passed to the target as the only parameters.

If the breakpoint is successfully removed, the target will respond with a RM_BKPT_ACK message. If no known breakpoint exists at that address, the target will respond with an ERROR message.

Message 7 (7h):
BKPT_STAT (Breakpoint Status)

Message

```
#define BKPT_STAT 7

struct bkpt_stat_msg_t {
         INT32    code; /* 7 */
         INT32    length;
         INT32    memory_space;
         ADDR32   bkpt_addr;
         };
```

where:

memory_space Is the address space of the breakpoint.

bkpt_address Is the address of the breakpoint.

Direction

Host-to-target

Acknowledgement

BKPT_STAT_ACK (on page 5–46)

Description

This message is used to request the status of a breakpoint from the target. The memory space and address of the breakpoint are passed to the target as the only parameters.

If the breakpoint exists, the target will respond with a BKPT_STAT_ACK message. If no known breakpoint exists at that address, the target will respond with an ERROR message.

This primitive typically is used to check the pass count of a breakpoint.

Message 8 (8h):
COPY (Copy Data)

Message

```
#define COPY 8

struct copy_msg_t {
         INT32    code; /* 8 */
         INT32    length;
         INT32    source_space;
         ADDR32   source_addr;
         INT32    dest_space;
         ADDR32   dest_addr;
         INT32    count;
         INT32    size;
         };
```

where:

source_space	Specifies the memory space of the source data.
source_addr	Is the address in this memory space of the data to be copied.
dest_space	Specifies the memory space for the destination of the copy operation.
dest_addr	Specifies the address for the destination of the copy operation.
count	Is a count of the number of objects to be copied from the source to the destination.
size	Specifies the size of the object in bytes to be copied (1 = byte, 2 = half word, and 4 = full word).

Direction

Host-to-target

Acknowledgement

COPY_ACK (on page 5–47)

Description

The COPY message is used to request that a block of memory be copied from one memory location to another. The source and destination do not have to reside in the same memory space.

This operation could be implemented using other primitives, but at the cost of host-to-target bandwidth. A COPY primitive has been included for efficiency.

Some or all of the source may not be readable, or some or all of the destination may not be writable. In either case, an ERROR message will be returned. No assumptions should be made by the host as to the amount of data copied by an unsuccessful operation.

Message 9 (9h):
FILL (Fill Memory)

Message

```
#define FILL 9

struct fill_msg_t {
         INT32    code; /* 9 */
         INT32    length;
         INT32    memory_space;
         ADDR32   start_addr;
         INT32    fill_count;
         INT32    byte_count;
         BYTE     fill_data[];
         };
```

where:

memory_space	Specifies the memory space where the FILL message will write blocks of memory.
start_addr	Specifies the beginning address of the blocks of memory to be filled.
fill_count	Specifies the number of bytes to be filled. Note that *fill_count* is not necessarily an even multiple of *byte_count*.
byte_count	Specifies the number of bytes in the string.
fill_data	Is a 32-bit value.

Direction

Host-to-target

Acknowledgement

FILL_ACK (on page 5–48)

Description

This primitive is used to fill blocks of memory. This message could have been built from other primitives, but a separate primitive was defined for the sake of efficiency.

The FILL message writes a block of memory in *memory_space* beginning at *start_addr* with copies of the byte string *fill_data*[]. The number of bytes in this string is given by *byte_count*. The number of bytes to be filled is given by *fill_count*. Note that *fill_count* is not necessarily an even multiple of *byte_count*.

This message represents a general form of pattern filling. Bytes may be filled in by setting *byte_count* to 1, and having a single element in the *fill_data* array. Thirty-two bit words may be filled by setting the *byte_count* to 4 and placing a 32-bit value in the *fill_data* array. Other more complicated fill patterns are possible with the FILL primitive.

The monitor may not have write access to some or all of the memory specified by the FILL message. In this case, the FILL message returns an ERROR message. No assumptions may be made by the host as to the value of memory locations involved in an unsuccessful FILL.

Message 10 (Ah): INIT (Initialize Target)

Message

```
#define INIT 10

struct init_msg_t {
         INT32    code; /* 10 */
         INT32    length;
         ADDR32   text_start;
         ADDR32   text_end;
         ADDR32   data_start;
         ADDR32   data_end;
         ADDR32   entry_point;
         INT32    mem_stack_size;
         INT32    reg_stack_size;
         ADDR32   arg_start;
         INT32    os_control;
         ADDR32   highmem;
         };
```

where:

text_start	Specifies the start address in instruction memory of the code that has been loaded for execution. This parameter is derived from the most recently loaded COFF file.
text_end	Specifies the end address in instruction memory of the code that has been loaded for execution. This parameter is derived from the most recently loaded COFF file.
data_start	Specifies the start in data memory of the data. This parameter is derived from the most recently loaded COFF file.
data_end	Specifies the end in data memory of the data. This parameter is derived from the most recently loaded COFF file.
entry_point	Is the entry point of the code. This parameter is derived from the most recently loaded COFF file.
mem_stack_size	This parameter may be useful to the target, but the target is under no obligation to use the value.

reg_stack_size This parameter may be useful to the target, but the target is under no obligation to use the value.

arg_start Is an address in data memory pointing to the command-line parameters. These parameters are stored as an array of pointers to strings. This array is terminated by a null pointer. This array, and the associated strings, typically are loaded into data memory by the host.

os_control Is a 32-bit coded value that is interpreted by the HIF kernel of **osboot** during the warm-start process. See the **osboot** manual for more information.

highmem Specifies the starting address for the register stack in memory. This is interpreted by the HIF kernel of **osboot** during the warm-start process. See the **osboot** manual for more information.

Direction

Host-to-target

Acknowledgement

INIT_ACK (on page 5–49)

Description

The INIT message is used to provide run-time information for the downloaded application program.

Message 11 (Bh):
GO (Execute Code)

Message

```
#define GO 11

struct go_msg_t {
         INT32    code; /* 11 */
         INT32    length;
         };
```

Direction

Host-to-target

Acknowledgement

HALT (on page 5–50), or any target-to-host operating-system message

Description

The GO message is used to initiate the execution of a piece of code. The message has no parameters. Code will begin executing according to the preset state of the target processor.

When execution is complete, a HALT message will be returned to the host.

In some cases, it may be necessary for the host to terminate the execution of the target code prematurely. In this case, a BREAK message may be sent before receipt of the HALT message.

Message 12 (Ch): STEP (Step Execution)

Message

```
#define STEP 12

struct step_msg_t {
         INT32    code; /* 12 */
         INT32    length;
         INT32    count;
         };
```

where:

count	Defines the number of instructions to be executed in the step. A *count* of 1 corresponds to the execution of a single instruction. Counts greater than 1 refer to corresponding step sizes. Counts of 0 or less have no meaning.

Direction

Host-to-target

Acknowledgement

HALT (on page 5–50)

Description

This message is used to step through a program. When the stepping is complete, a STEP_ACK message is returned to the host. Note that when stepping through multiple instructions, no trace information is returned. The instructions actually executed will not be known to the host. If this information is desired, a series of single steps must be executed by the host.

Message 13 (Dh): BREAK (Stop Execution)

Message

```
#define BREAK 13

struct break_msg_t {
         INT32    code; /* 13 */
         INT32    length;
         };
```

Direction

Host-to-target

Acknowledgement

HALT (on page 5–50)

Description

The BREAK message is used to stop the execution of running code. This message has no parameters. The *length* field will always be set to 0. When the execution of the target code has been successfully halted, a HALT message is returned.

Message 33 (21h): CONFIG (Target Configuration)

Message

```
#define CONFIG 33

struct config_msg_t {
         INT32    code; /* 33 */
         INT32    length;
         INT32    processor_id;
         INT32    version;
         ADDR32   I_mem_start;
         INT32    I_mem_size;
         ADDR32   D_mem_start;
         INT32    D_mem_size;
         ADDR32   ROM_start;
         INT32    ROM_size;
         INT32    max_msg_size;
         INT32    max_bkpts;
         INT32    coprocessor;
         INT32    os_version;
         };
```

where:

processor_id	Is a number that describes the target processor. It should contain the processor identification number (PID) of the target processor.
version	Specifies the version number of the MiniMON29K target monitor software.
I_mem_start	Specifies the starting address of the instruction memory.
I_mem_size	Specifies the size of instruction memory in bytes.
D_mem_start	Specifies the starting address of the data memory.
D_mem_size	Specifies the size of data memory in bytes.
ROM_start	Specifies the starting address of the ROM.
ROM_size	Specifies the size of ROM in bytes.
max_msg_size	Specifies the size of the target message buffer in bytes. This parameter defines the largest message that the target will accept.

max_bkpts	Specifies the maximum number of breakpoints supported on the target.
coprocessor	Specifies the system coprocessor. A value of –1 means that no coprocessor is present. The only coprocessor supported by the MiniMON29K product is the Am29027 coprocessor. If the Am29027 coprocessor is present, this field has a value of 0. Only one coprocessor per system is supported.
os_version	Is the target OS version number.

Direction

Target-to-host

Requestor

CONFIG_REQ (on page 5–17)

Description

This message returns configuration information from the target. If the information concerning a particular parameter is not available, the parameter should be set to –1. This message is sent in response to the CONFIG_REQ message from the host.

Other system-specific parameters may be added to the end of this parameter list. The host is expected to recognize CONFIG messages of various lengths. The extra parameters at the end of this list of standard configuration parameters are application specific, and will not be interpreted by the standard host interface tools.

Message 34 (22h): STATUS (Target Status)

Message

```
#define STATUS 34

struct status_msg_t {
         INT32    code; /* 34 */
         INT32    length;
         INT32    msgs_sent;
         INT32    msgs_received;
         INT32    errors;
         INT32    bkpts_hit;
         INT32    bkpts_free;
         INT32    traps;
         INT32    fills;
         INT32    spills;
         INT32    cycles_hi;
         INT32    cycles_lo;
         INT32    reserved;
         };
```

where:

msgs_sent — Specifies the number of messages sent by the target to the host.

msgs_received — Specifies the number of messages received by the target from the host.

errors — Specifies the number of error messages sent from the target to the host.

bkpts_hit — Specifies the number of breakpoints hit by the target. Breakpoints encountered in the context of pass counts are also considered breakpoints hit. For instance, a breakpoint with a pass count of 3 will account for three breakpoints hit in the parameter.

bkpts_free	Specifies the number of available breakpoints on the target. This parameter assumes that there is a limited number of breakpoints managed by the target. If there is no limit to the number of breakpoints available on the target, this parameter should be set to a sufficiently large number.
traps	Specifies a count of the total number of traps taken by the user code.
fills	Specifies a count of the total number of fill traps taken by the user code. Note that a fill trap will increment the count of both the *traps* parameter and the *fills* parameter.
spills	Specifies a count of the total number of spill traps taken by the user code. Note that a spill trap will increment the count of both the *traps* parameter and the *spills* parameter.
cycles_hi	Specifies the high word of the count of the total number of cycles of user code executed on the target. This number is reset to 0 each time the target processor is reset.
cycles_lo	Specifies the low word of the count of the total number of cycles of user code executed on the target. This number is reset to 0 each time the target processor is reset.
reserved	Is reserved for future use.

Direction

Target-to-host

Requestor

STATUS_REQ (on page 5–18)

Description

This message returns run-time status information from the target. If the information concerning a particular parameter is not available, that parameter will be set to –1. This message is sent in response to the STATUS_REQ message from the host.

Like the CONFIG message, other parameters may be added to the end of this parameter list. The host is expected to recognize STATUS messages of various lengths. The extra parameters at the end of this list of standard status parameters are application specific, and will not be interpreted by the standard host interface tools.

Message 35 (23h): READ_ACK (Read Memory)

Message

```
#define READ_ACK 35

struct read_ack_msg_t {
        INT32    code; /* 35 */
        INT32    length;
        INT32    memory_space;
        ADDR32   address;
        INT32    byte_count;
        BYTE     data[];
        };
```

where:

memory_space	Specifies the memory space of the returned data. This should match the *memory_space* parameter in the READ_REQ message.
address	Specifies the address of the returned data. This should match the *address* parameter in the READ_REQ message.
byte_count	Specifies the number of bytes in the data array that is returned in the message. This value should also match the *byte_count* specified in the READ_REQ message. Note that this parameter is somewhat redundant. The *byte_count* could be derived from the *length* parameter in the message header. It is included for convenience in reading the data array.
data	Is an array of 8-bit bytes. Data is returned as bytes because this is the smallest accessible data element on most machines. It is the responsibility of the host code to properly interpret the raw data returned by this message.

Direction

Target-to-host

Requestor

READ_REQ (on page 5–19)

Description

This message returns data requested by a READ_REQ message. It may not be possible to fulfill the READ_REQ because of a lack of message buffer space or an inability to access the memory. In these cases, an ERROR message is returned.

Message 36 (24h): WRITE_ACK (Data Written)

Message

```
#define WRITE_ACK 36

struct write_ack_msg_t {
         INT32    code; /* 36 */
         INT32    length;
         INT32    memory_space;
         ADDR32   address;
         INT32    byte_count;
         };
```

where:

memory_space Specifies the memory space of the written data. This should match the *memory_space* parameter in the WRITE_REQ message.

address Specifies the address of the written data. This should match the *address* parameter in the WRITE_REQ message.

byte_count Is the size of the data array in 8-bit bytes.

Direction

Target-to-host

Requestor

WRITE_REQ (on page 5–21)

Description

This message is sent in acknowledgement of a successful WRITE_REQ operation.

Message 37 (25h): BKPT_SET_ACK (Breakpoint Set)

Message

```
#define BKPT_SET_ACK 37

struct bkpt_set_ack_msg_t {
         INT32    code; /* 37 */
         INT32    length;
         INT32    memory_space;
         ADDR32   address;
         INT32    pass_count;
         };
```

where:

memory_space Is the address space where the breakpoint is to be set. In most cases, this will be the instruction memory of the system.

address Is the address of the breakpoint.

pass_count Specifies the number of times the breakpoint must be encountered before control is passed from the application to the monitor. A *pass_count* of 1 means that a break should occur the next time the instruction at this address is executed.

Direction

Target-to-host

Requestor

BKPT_SET (on page 5–23)

Description

This message acknowledges the successful setting of a breakpoint.

Message 38 (26h): BKPT_RM_ACK (Breakpoint Removed)

Message

```
#define BKPT_RM_ACK 38

struct bkpt_rm_ack_msg_t {
         INT32    code; /* 38 */
         INT32    length;
         INT32    memory_space;
         ADDR32   address;
         };
```

where:

memory_space	Specifies the memory space of the breakpoint that was removed.
address	Specifies the address of the breakpoint that was removed.

Direction

Target-to-host

Requestor

BKPT_RM (on page 5–25)

Description

This message acknowledges the successful removal of a breakpoint. The parameters for this message will have the same values as those passed to the target in the RM_BKPT message.

Message 39 (27h): BKPT_STAT_ACK (Breakpoint Status)

Message

```
#define BKPT_STAT_ACK 39

struct bkpt_stat_ack_msg_t {
         INT32    code; /* 39 */
         INT32    length;
         INT32    memory_space;
         ADDR32   address;
         INT32    pass_count;
         };
```

where:

memory_space	Specifies the same values passed to the target in the BKPT_STAT message.
address	Specifies the same values passed to the target in the BKPT_STAT message.
pass_count	Specifies the number of times the breakpoint must be encountered before control is passed from the application to the monitor. A *pass_count* of 1 means that a break should occur the next time the instruction at this address is executed.

Direction

Target-to-host

Requestor

BKPT_STAT (on page 5–26)

Description

This message is sent in response to the BKPT_STAT message. This message returns the status of the breakpoint at the requested address. The primary use of this message is to inspect the *pass_count* of a breakpoint.

Message 40 (28h): COPY_ACK (Data Copied)

Message

```
#define COPY_ACK 40

struct copy_ack_msg_t {
         INT32    code; /* 40 */
         INT32    length;
         INT32    source_space;
         ADDR32   source_addr;
         INT32    dest_space;
         ADDR32   dest_addr;
         INT32    byte_count;
         };
```

where:

source_space	Specifies the memory space of the source data.
source_addr	Is the address in this memory space of the data to be copied.
dest_space	Specifies the memory space for the destination of the copy operation.
dest_addr	Specifies the address for the destination of the copy operation.
byte_count	Is a count of the number of bytes to be copied from the source to the destination.

Direction

Target-to-host

Requestor

COPY (on page 5–27)

Description

The COPY_ACK message acknowledges that the operation requested by COPY has completed successfully. The COPY_ACK should return the same parameters sent by the COPY message.

Message 41 (29h): FILL_ACK (Memory Filled)

Message

```
#define FILL_ACK 41

struct fill_ack_msg_t {
         INT32    code; /* 41 */
         INT32    length;
         INT32    memory_space;
         ADDR32   start_addr;
         INT32    fill_count;
         INT32    byte_count;
         };
```

where:

memory_space	Specifies the memory space where the FILL message will write blocks of memory.
start_addr	Specifies the beginning address of the blocks of memory to be filled.
fill_count	Specifies the number of bytes to be filled. Note that *fill_count* is not necessarily an even multiple of *byte_count*.
byte_count	Specifies the number of bytes in the string.

Direction

Target-to-host

Requestor

FILL (on page 5–29)

Description

The FILL_ACK message acknowledges that a FILL message has been successfully executed.

FILL_ACK should return the first four parameters sent by the FILL message. Because this message serves only as an acknowledgement, the fill pattern sent by the FILL message is not returned.

Message 42 (2Ah):
INIT_ACK (Target Initialized)

Message

```
#define INIT_ACK 42

struct init_ack_msg_t {
         INT32    code; /* 42 */
         INT32    length;
         };
```

Direction

Target-to-host

Requestor

INIT (on page 5–31)

Description

The INIT_ACK message acknowledges that an INIT message has been received by the target.

Message 43 (2Bh): HALT (Execution Halted)

Message

```
#define HALT 43

struct halt_msg_t {
         INT32    code; /* 43 */
         INT32    length;
         INT32    memory_space;
         ADDR32   pc0;
         ADDR32   pc1;
         INT32    trap_number;
         };
```

where:

pc0	Contains the value of the program counter register **pc0**.
pc1	Contains the value of the program counter register **pc1**.
trap_number	Is the value of the trap number that caused the halt.

Direction

Target-to-host

Requestor

BREAK (on page 5–35), GO (on page 5–33), RESET (on page 5–16) or STEP (on page 5–34)

Description

The HALT message is sent to the host any time control is returned to the monitor. It is sent in response to a GO or a STEP message.

Message 63 (3Fh): ERROR (Error Detected)

Message

```
#define ERROR 63

struct error_msg_t {
         INT32    code; /* 63 */
         INT32    length;
         INT32    error_code;
         };
```

where:

error_code Is a 32-bit integer containing an error code. This error code describes the error encountered when attempting to execute a host command.

Direction

Target-to-host

Requestor

Any host-to-target message

Description

This message is returned to the host whenever an error is encountered. An ERROR message may be returned from any host request. This message sends a single parameter.

Operating-System Messages

Messages 64 through 127 are classified as operating-system messages. These messages are transmitted between the host and the application/operating system running on the target. In the default configuration of MiniMON29K, the HIF kernel of **osboot** transmits and receives the operating-system messages on the target.

- Messages 64 through 95 are sent from the host to the target.
- Messages 96 through 127 are sent from the target to the host.

The operating-system messages are listed on the following pages, in numerical order. See page 5–15 for the debug messages.

Message 64 (40h): HIF_CALL_RTN (HIF_CALL Return)

Message

```
#define HIF_CALL_RTN 64

struct hif_call_rtn_msg_t {
         INT32    code;  /* 64 */
         INT32    length;
         INT32    service_number;
         INT32    gr121;
         INT32    gr96;
         INT32    gr97;
         };
```

Direction

Host-to-target

Requestor

HIF_CALL (on page 5–59)

Description

The HIF_CALL_RTN message is used by **montip** to return the results of the requested HIF system call to the HIF kernel that requested it. The *service_number* field contains the HIF service number of the requested operation. (See AMD's host interface specification for more details.) The fields gr121, gr96, and gr97 contain the results of the requested operation according to the service requested.

Message 65 (41h): CHANNEL0 (Data at Channel 0)

Message

```
#define CHANNEL0 65

struct channel0_msg_t {
         INT32    code; /* 65 */
         INT32    length;
         BYTE     data;
         };
```

Direction

Host-to-target

Acknowledgement

CHANNEL0_ACK (on page 5–60)

Description

The CHANNEL0 message is used by the host to send a single byte to the target. This is typically a key pressed on the keyboard. This provides “raw” keyboard input.

This message is acknowledged by CHANNEL0_ACK.

Message 66 (42h): CHANNEL1_ACK (Channel 1 Ack)

Message

```
#define CHANNEL1_ACK 66

struct channel1_ack_msg_t {
        INT32    code; /* 66 */
        INT32    length;
        INT32    nbytes;
        };
```

Direction

Host-to-target

Requestor

CHANNEL1 (on page 5–61)

Description

The CHANNEL1_ACK message is used by the host to acknowledge that a CHANNEL1 message has been read and processed. The CHANNEL1_ACK message returns to the standard output device the number of bytes successfully written in the *nbytes* parameter.

Message 67 (43h): CHANNEL2_ACK (Channel 2 Ack)

Message

```
#define CHANNEL2_ACK 67

struct channel2_ack_msg_t {
         INT32    code; /* 67 */
         INT32    length;
         INT32    nbytes;
         };
```

Direction

Host-to-target

Requestor

CHANNEL2 (on page 5–62)

Description

The CHANNEL2_ACK message is used by the host to acknowledge that a CHANNEL2 message has been read and processed. The CHANNEL2_ACK message returns to the standard output device the number of bytes successfully written in the *nbytes* parameter.

Message 68 (44h): STDIN_NEEDED_ACK (Standard Input Needed)

Message

```
#define STDIN_NEEDED_ACK 68

struct stdin_needed_ack_msg_t {
         INT32    code;  /* 68 */
         INT32    length;
         BYTE     data;
};
```

Direction

Host-to-target

Requestor

STDIN_NEEDED (on page 5–63)

Description

The STDIN_NEEDED_ACK message is sent in response to a request from the target for input from the standard input device, i.e., terminal. This message is used when the standard input mode is in synchronous and blocking mode.

The length field of the message contains the number of input characters that follow. The data field has the first input character.

Message 69 (45h): STDIN_MODE_ACK (Standard Input Mode)

Message

```
#define STDIN_MODE_ACK 69

struct stdin_mode_ack_msg_t {
         INT32    code;  /* 69 */
         INT32    length;
         INT32    mode;
};
```

Direction

Host-to-target

Requestor

STDIN_MODE (on page 5–64)

Description

When the target sends a STDIN_MODE message to change the standard input mode, the host sends the STDIN_MODE_ACK message in response. The *mode* field of the message contains the previous input mode. AMD’s host interface specification enumerates the mode values for the different input modes.

Message 96 (60h):
HIF_CALL (HIF Call)

Message

```
#define HIF_CALL 96

struct hif_call_msg_t {
        INT32    code; /* 96 */
        INT32    length;
        INT32    service_number; /* gr121 */
        INT32    lr2;
        INT32    lr3;
        INT32    lr4;
        };
```

Direction

Target-to-host

Acknowledgement

HIF_CALL_RTN (on page 5–53)

Description

The HIF_CALL message is used by the HIF kernel of **osboot** to request a HIF operating-system service from the host. The host should perform the requested action (if possible) and send the results in a HIF_CALL_RTN message.

Message 97 (61h): CHANNEL0_ACK (Channel 0 Acknowledgement)

Message

```
#define CHANNEL0_ACK 97

struct channel0_ack_msg_t {
         INT32    code; /* 97 */
         INT32    length;
         };
```

Direction

Target-to-host

Requestor

CHANNEL0 (on page 5–54)

Description

The CHANNEL0_ACK message is used by the target to acknowledge that the byte sent by the CHANNEL0 message has been received. This message has no parameters.

Message 98 (62h): CHANNEL1 (Write Channel 1)

Message

```
#define CHANNEL1 98

struct channel1_msg_t {
         INT32    code; /* 98 */
         INT32    length;
         BYTE     data[];
         };
```

Direction

Target-to-host

Acknowledgement

CHANNEL1_ACK (on page 5–55)

Description

The CHANNEL1 message is used by the target to write an array of bytes to the host standard output device.

Message 99 (63h): CHANNEL2 (Write Channel 2)

Message

```
#define CHANNEL2 99

struct channel2_msg_t {
         INT32    code; /* 99 */
         INT32    length;
         BYTE     data[];
         };
```

Direction

Target-to-host

Acknowledgement

CHANNEL2_ACK (on page 5–56)

Description

The CHANNEL2 message is used by the target to write an array of bytes to the host standard error device.

Message 100 (64h): STDIN_NEEDED (Standard Input Needed)

Message

```
#define STDIN_NEEDED 100

struct stdin_needed_msg_t {
         INT32    code;  /* 100 */
         INT32    length;
         INT32    nbytes;
         };
```

Direction

Target-to-host

Acknowledgement

STDIN_NEEDED_ACK (on page 5–57)

Description

When the input mode is synchronous and blocking, the operating system or application program running on the target system sends a STDIN_NEEDED message to the host to request user input. The *nbytes* field contains the maximum number of bytes requested at this time. The host waits for input to be available and returns the input data using the STDIN_NEEDED_ACK message.

Message 101 (65h): STDIN_MODE (Standard Input Mode)

Message

```
#define STDIN_MODE 101

struct stdin_mode_msg_t {
         INT32    code;  /* 101 */
         INT32    length;
         INT32    mode;
         };
```

Direction

Target-to-host

Acknowledgement

STDIN_MODE_ACK (on page 5–58)

Description

This message is sent by the target to request a change in the input mode on the standard input device. The *mode* field contains the code for the input mode requested. AMD's host interface specification enumerates the mode values for different input modes. The host sends a STDIN_MODE_ACK message which contains the previous input mode.

Appendix A

MONTIP Error Messages

The **montip** error messages are listed on the following page in order of error number. However, note that the error message may appear differently as the format of the error messages varies depending on the DFE being used.

MONTIP Error Messages

0 MONNoError: “No Error.”

1 MONErrCantSendMsg: “Could not send message to target.”

2 MONErrCantRecvMsg: “Did not receive the correct ACK from target.”

3 MONErrCantLoadROMfile: “Can’t load ROM file.”

4 MONErrCantInitMsgSystem: “Can’t initialize the message system.”

5 MONErrCantBreakInROM: “Can’t set breakpoint in ROM.”

6 MONErrCantResetComm: “Can’t reset communication channel.”

7 MONErrCantAllocBufs: “Can’t reallocate message buffers.”

8 MONErrUnknownBreakType: “Breakpoint type requested is not recognized.”

9 MONErrNoAck: “No ACK from target—timed out.”

10 MONErrNoSynch: “Timed out synching. No response from target.”

11 MONErrCantOpenCoff: “Cannot open ROM file.”

12 MONErrCantWriteToMem: “Cannot write to memory while downloading ROM file.”

13 MONErrAbortAborted: “Ctrl-C aborted previous Ctrl-C processing.”

14 MONErrNullConfigString: “Null configuration string specified for connection.”

15 MONErrNoTargetType: “No target type specified for connection.”

16 MONErrOutofMemory: “Out of memory.”

17 MONErrErrorInit: “Error on target—trying to initialize process.”

18 MONErrErrorRead: “Error on target—trying to read.”

19 MONErrErrorWrite: “Error on target—trying to write.”

20 MONErrErrorCopy: “Error on target—trying to do copy.”

21 MONErrErrorSetBreak: “Error on target—trying to set breakpoint.”

22 MONErrErrorStatBreak: “Error on target—trying to query breakpoint.”

23 MONErrErrorRmBreak: “Error on target—trying to remove breakpoint.”

24 MONErrConfigInterrupt: “User interrupt signal received; aborting synch.”

25 MONErrNoConfig: “Couldn’t get target config after reset. Try again.”

26 MONErrMsgInBuf: “Message received from target waiting in buffer.”

27 MONErrUnknownTIPCmd: “Unknown MONTIP command; exiting TIP mode.”

Appendix B

MiniMON29K Target Message System

The code for the MiniMON29K Target Message system, contained in the **msg.s** file, is shown on the following pages.

msg.s File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This is the Message System of MiniMON29K.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
       .file   ”msg.s”

       .ident  ”@(#)msg.s     1.3 93/07/06 18:14:28, Srini, AMD”

       ; MiniMON29K R 1.1 Version    don’t know
       ; MiniMON29K R 2.0 Version    0x10
       ; MiniMON29K R 2.1 Version    0x11
       ; MiniMON29K R 3.0 Version    0x12
       .equ    MSG_VERSION,   0x12

       .equ    MSG_RBUF_SIZE, 2048

       .extern dbg_V_msg              ; Debug core’s message handler
       .extern os_V_msg               ; OS message handler

       .extern msg_write_p            ; function to write out a message
       .extern msg_wait_for_p         ; ptr to function that waits for a msg

       .extern msg_initcomm

       .global _msg_version           ; version of msg sys and comm drivers

       .global _msg_sbuf_p            ; address of the message to send
       .global _msg_lastsent_p        ; address of last msg sent
       .global _msg_next_p            ; next char to send
       .global _msg_rbuf              ; message receive buffer

       .global _msg_init              ; init msg sys at cold start
       .global _msg_send              ; send valid msg to montip

       .global msg_V_arrive           ; get here on receiving a message
       .global _msg_wait_for          ; wait for message to arrive
```

msg.s File continued

```
        .macro  MSG_SAVE_GLOB
        const   gr4, _msg_save_glob
        consth  gr4, _msg_save_glob
        store   0, 0, gr96, gr4               ; gr96
        const   gr96, _msg_save_glob+4
        consth  gr96, _msg_save_glob+4
        store   0, 0, gr97, gr96      ; gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96      ; gr98
        .endm

        .macro  MSG_RESTORE_GLOB
        const   gr96, _msg_save_glob+4
        consth  gr96, _msg_save_glob+4
        load    0, 0, gr97, gr96      ; gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96      ; gr98
        const   gr96, _msg_save_glob
        consth  gr96, _msg_save_glob
        load    0, 0, gr96, gr96      ; gr96
        .endm

        .sect   msg_data, bss
        .use    msg_data
_msg_sbuf_p:           .block  1*4
_msg_rbuf:             .block  MSG_RBUF_SIZE
_msg_version:          .block  1*4
_msg_lastsent_p:       .block 1*4
_msg_next_p:           .block 1*4
_msg_save_glob:               .block  3*4            ; gr96-gr98
_msg_save_loc:         .block  3*4            ; lr0-lr2
_msg_v_save:           .block  3*4            ; gr96-gr98
_msg_send_save:               .block  6*4            ; lr0-lr4

        .text
; ---------------------------------------------------------------------------
; initialize msg system data structures.
; msg system initialization. The actual device depends on the target system
; and is initialized by msg_initcomm function.
_msg_init:
        MSG_SAVE_GLOB

        const   gr96, _msg_lastsent_p
        consth  gr96, _msg_lastsent_p
        const   gr97, 0
        store   0, 0, gr97, gr96      ; init last msg address
```

msg.s File continued

```
const   gr96, _msg_sbuf_p
consth  gr96, _msg_sbuf_p
store   0, 0, gr97, gr96      ; clear semaphore

const   gr96, _msg_next_p
consth  gr96, _msg_next_p
const   gr97, _msg_rbuf
consth  gr97, _msg_rbuf
store   0, 0, gr97, gr96      ; next char to send pointer

const   gr96, _msg_save_loc
consth  gr96, _msg_save_loc
store   0, 0, lr0, gr96       ; save lr0

const   gr96, msg_initcomm    ; returns version number
consth  gr96, msg_initcomm
calli   lr0, gr96             ; initialize comm interface
nop

; gr96 has the comm_version number.
; initialize msg_version with msg_version|comm_version
sll     gr96, gr96, 8         ; driver version
or      gr96, gr96, MSG_VERSION       ; append msg sys version
const   gr97, _msg_version
consth  gr97, _msg_version
store   0, 0, gr96, gr97      ; store for use by debug core

; restore lr0
const   gr96, _msg_save_loc
consth  gr96, _msg_save_loc
load    0, 0, lr0, gr96       ; restore lr0
MSG_RESTORE_GLOB

jmpi    lr0
nop
```

```
; ------------------------------------------------------------ MSG_V_ARRIVE
msg_V_arrive:
       const   gr4, _msg_rbuf        ; determine class, (first entry)
       consth  gr4, _msg_rbuf        ; determine class, (first entry)
       load    0, 0, gr4, gr4
       cpgeu   gr4, gr4, 64
       jmpt    gr4, os_msg
dbgcore_msg:
       const   gr4, dbg_V_msg
       consth  gr4, dbg_V_msg
       jmpi    gr4                   ;jmpi to dbg_V_msg
        nop
os_msg:
       const   gr4, os_V_msg
       consth  gr4, os_V_msg
       jmpi    gr4                   ;jmpi to os_V_msg
        nop

;------------------------------------------------------------- MSG_WAIT_FOR
; This function is used to indicate if the receive message
; buffer contains a vaild message. The return value is –1
; if the buffer is valid, and 0 if invalid.
; With a poll driven serial driver, msg_wait_for() should
; not return until the buffer contains a message for processing.
;
_msg_wait_for:
       ;
       ; save lr0
       ;
       const   gr4, _msg_save_loc
       consth  gr4, _msg_save_loc
       store   0, 0, lr0, gr4        ; save lr0

       const   gr96, msg_wait_for_p
       consth  gr96, msg_wait_for_p
       load    0, 0, gr96, gr96      ; get function address

       calli   lr0, gr96
       nop

       const   lr0, _msg_save_loc
       consth  lr0, _msg_save_loc
       load    0, 0, lr0, lr0        ; restore lr0

       jmpi    lr0                   ; return
       nop
```

msg.s File continued

```
;-------------------------------------------------------------- MSG_SEND
_msg_send:
; Send the message pointed to by lr2.
; return success (–1) or failure (0) in gr96 to caller.

       ; check msg send semaphore
       const   gr96, _msg_sbuf_p
       consth  gr96, _msg_sbuf_p
       load    0, 0, gr96, gr96      ; read semaphore
       cpeq    gr96, gr96, 0         ; compare with zero
       jmpfi   gr96, lr0             ; if not zero, return failure
       constn  gr96, –1              ; –1 for failure

       ; update semaphore with address of message to send in lr2
       const   gr96, _msg_sbuf_p
       consth  gr96, _msg_sbuf_p
       store   0, 0, lr2, gr96       ; update msg send semaphore

       const   gr96, _msg_lastsent_p
       consth  gr96, _msg_lastsent_p
       store   0, 0, lr2, gr96       ; update msg last sent pointer

       ; backup some registers for temporary use.
       const   gr96, _msg_send_save
       consth  gr96, _msg_send_save
       store   0, 0, lr0, gr96       ; save lr0
       add     gr96, gr96, 4
       store   0, 0, lr1, gr96       ; save lr1
       add     gr96, gr96, 4
       store   0, 0, lr2, gr96       ; save lr2
       add     gr96, gr96, 4
       store   0, 0, lr3, gr96       ; save lr3

       ; send the message calling the device write function.
       ; offset 0 in write_table is dedicated for debug core.
       ; lr2 = pointer to message
       ; lr3 = total bytes in message.
       add     lr3, lr2, 4
       load    0, 0, lr3, lr3        ; get msg length value
       add     lr3, lr3, 8           ; add msg header size
       const   gr96, msg_write_p
       consth  gr96, msg_write_p
       load    0, 0, gr96, gr96      ; get msg_write function.
       calli   lr0, gr96             ; call msg_write
       nop
```

```
;
;
const   gr96, _msg_send_save
consth  gr96, _msg_send_save
load    0, 0, lr0, gr96       ; restore lr0
add     gr96, gr96, 4
load    0, 0, lr1, gr96       ; restore lr1
add     gr96, gr96, 4
load    0, 0, lr2, gr96       ; restore lr2
add     gr96, gr96, 4
load    0, 0, lr3, gr96       ; restore lr3

jmpi    lr0
const   gr96, 0               ; return success
```

Appendix C

Target Message Drivers

At the time of publication, the code for the target message drivers was in the files with the following names:

- **eb29khw.s** file: Code for the EB29K target message driver
- **eb030hw.s** file: Code for the EB29030 target message driver
- **scc8530.s** and **ez030hw.s** files: Code for the EZ-030 target message driver
- **scc200.s** and **sa200hw.s** files: Code for the SA-29200 and SA-29205 target message driver. The contents of these two files also are printed on the following pages.

scc200.s File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This module implements the routines for Am29200 SCC on chip.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
       .ident  ”@(#)scc200.s  1.7 93/11/01 09:11:20, Srini, AMD”

       .file   ”scc200.s”

       .include ”stats.ah”

       .equ    NACK_BIT, 0x1
       .equ    ACK_BIT, 0x2

       .extern UCLK           ; link time constant def in linker command file

       .ifndef BAUDRATE
       .equ    BAUDRATE,      9600
       .endif

       .global msg_scc200_init
       .global msg_scc200_write
       .global msg_scc200_wait_for    ; polled mode receive
       .global msg_scc200_tx_intr
       .global msg_scc200_rx_intr
       .global msg_ppi200_intr
       .global msg_lpt200_init

       .extern _msg_rbuf              ; start of receive buffer
       .extern _msg_next_p            ; message receive buffer pointer
       .extern _msg_lastsent_p        ; address of msg last sent
       .extern _msg_sbuf_p            ; message send semaphore

       .extern msg_V_arrive           ; virtual message interrupt vector
```

scc200.s File continued

```
        .bss
scc200_tmp_regs:        .block  4*4
intr_tmp_regs:          .block  4*4
poll_tmp_glob:          .block  4*4     ; gr97-gr99
poll_tmp_loc:           .block  3*4     ; lr0, lr2-lr3
nbytes_to_write:        .block 1*4
nextchar_p:             .block  1*4
ack_flag:               .block  1*4
nack_flag:              .block  1*4
firstmsg_flag:          .block  1*4
ack_msg_p:              .block  2*4
nack_msg_p:             .block  2*4

        .text
; ------------------------------------------------------------ MSG_SCC200_INIT
msg_scc200_init:
; gr96, gr97, gr98 are saved before calling this.
; Returns via lr0.
        ; initialize ack msg and nack msg structures.
        const   gr96, ack_msg_p
        consth  gr96, ack_msg_p
        constn  gr97, -1
        store   0, 0, gr97, gr96        ; -1
        add     gr96, gr96, 4
        const   gr97, 0
        store   0, 0, gr97, gr96        ; 0

        const   gr96, nack_msg_p
        consth  gr96, nack_msg_p
        constn  gr97, -1
        store   0, 0, gr97, gr96        ; -1
        add     gr96, gr96, 4
        store   0, 0, gr97, gr96        ; -1

        ; set the firstmsg_flag to true
        const   gr96, firstmsg_flag
        consth  gr96, firstmsg_flag
        constn  gr97, -1
        store   0, 0, gr97, gr96        ; firstmsg_flag = TRUE

        const   gr97, SPCT
        consth  gr97, SPCT
        const   gr96, 0
        store   0, 0, gr96, gr97                ; SPCT=0
```

scc200.s File continued

```
        ; compute baud rate
        ; bauddiv = UCLK/32/BAUDRATE – 1
        const   gr98, BAUDRATE
        consth  gr98, BAUDRATE
        const   gr96, UCLK
        consth  gr96, UCLK
        srl     gr96, gr96, 5                 ; / 32
        mtsr    q, gr96
        div0    gr97, 0
        .rep    31
        div     gr97, gr97, gr98
        .endr
        divl    gr97, gr97, gr98
        mfsr    gr97, q
        sub     gr96, gr97, 1                 ; bauddiv in gr96
        const   gr97, BAUD
        consth  gr97, BAUD
        store   0, 0, gr96, gr97              ; set BAUD

        const   gr96, 0x01030000              ; rx=intr mode, tx=intr mode
        consth  gr96, 0x01030000              ; word length=8 bits
        const   gr97, SPCT
        consth  gr97, SPCT
        store   0, 0, gr96, gr97             ; set rx,tx mode, wl=9bits,noparity

  .ifndef SERIAL_POLL
        const   gr96, 0x01030101              ; rx=intr mode, tx=intr mode
        consth  gr96, 0x01030101              ; word length=8 bits
        const   gr97, SPCT
        consth  gr97, SPCT
        store   0, 0, gr96, gr97             ; set rx,tx mode, wl=9bits,noparity
  .endif

        const   gr96, ICT
        consth  gr96, ICT
        const   gr97, (PPI|RXSI|RXDI|TXDI)
        consth  gr97, (PPI|RXSI|RXDI|TXDI)
        store   0, 0, gr97, gr96           ; reset serial port pending interrupts

        jmpi    lr0
        nop
```

```
; ----------------------------------------------------------- MSG_LPT200_INIT
msg_lpt200_init:
; gr96, gr97, gr98 are saved before calling this.
; Returns via lr0.
        const   gr96, PPCT
        consth  gr96, PPCT
        const   gr97, ((16<<TDELAYShift)|(1<<PPCT_MODEShift));
        consth  gr97, ((16<<TDELAYShift)|(1<<PPCT_MODEShift));
        store   0, 0, gr97, gr96      ; 8 bits, interrupt on char

        jmpi    lr0
        nop

; ------------------------------------------------------ MSG_SCC200_WRITE
msg_scc200_write:
; In interrupt mode, it sends out the first character and returns. In this
; mode it is called with interrupts disabled. Interrupts are enabled after
; this call returns.
; In polled mode, it loops until the entire message is written.
; Called from msg_send. return via lr0.
; lr2 – pointer to message
; lr3 = nbytes in message.
        const   gr4, scc200_tmp_regs
        consth  gr4, scc200_tmp_regs
        store   0, 0, gr96, gr4       ; backup gr96
        const   gr96, scc200_tmp_regs+4
        consth  gr96, scc200_tmp_regs+4
        store   0, 0, gr97, gr96      ; backup gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96      ; backup gr98

        ; set nextchar_p
        const   gr96, nextchar_p
        consth  gr96, nextchar_p
        store   0, 0, lr2, gr96       ; next char to send

        ; check the type of message, ack/nack have no checksum
        const   gr96, nbytes_to_write
        consth  gr96, nbytes_to_write
        load    0, 0, gr98, lr2
        jmpt    gr98, acknack_code
        add     gr97, lr3, 0
        add     gr97, lr3, 4          ; add checksum size
        store   0, 0, gr97, gr96      ; write nbytes to send
```

```
        ; compute checksum and append to end of message.
        add     gr96, lr2, 4
        load    0, 0, gr96, gr96        ; msg len
        add     gr96, gr96, 8           ; add msg size

        sub     gr96, gr96, 2
        const   gr98, 0                 ; initialize checksum
$1:
        load    0, 1, gr97, lr2
        add     gr98, gr98, gr97
        jmpfdec gr96, $1
        add     lr2, lr2, 1

        ; aapend at lr2
        srl     gr97, gr98, 24
        store   0, 1, gr97, lr2
        srl     gr97, gr98, 16
        and     gr97, gr97, 0xff
        add     lr2, lr2, 1
        store   0, 1, gr97, lr2
        srl     gr97, gr98, 8
        and     gr97, gr97, 0xff
        add     lr2, lr2, 1
        store   0, 1, gr97, lr2
        and     gr97, gr98, 0xff
        add     lr2, lr2, 1
        store   0, 1, gr97, lr2

        ; Start sending out the message. This layer does not buffer
        ; the message. Instead it relies on the message remaining
        ; there until it is sent. A semaphore msg_send_p is cleared
        ; when the message is sent.

        ; wait for transmit holding register to empty.
        const   gr96, SPST
tx_loop:
        consth  gr96, SPST
        load    0, 0, gr96, gr96        ; read status
        sll     gr96, gr96, (31 – THREShift)
        jmpf    gr96, tx_loop
        const   gr96, SPST
```

```
      ; get character from nextchar_p
      const   gr96, nextchar_p
      consth  gr96, nextchar_p
      load    0, 0, gr97, gr96
      load    0, 1, gr98, gr97      ; get character to send
      add     gr97, gr97, 1         ; update
      store   0, 0, gr97, gr96      ; nextchar_p++

      ; stuff character
      const   gr96, SPTH
      consth  gr96, SPTH
      store   0, 0, gr98, gr96      ; put char

      ; decrement nbytes_to_write
      const   gr96, nbytes_to_write
      consth  gr96, nbytes_to_write
      load    0, 0, gr97, gr96
      sub     gr97, gr97, 1
      store   0, 0, gr97, gr96      ; nbytes_to_write––

 .ifdef SERIAL_POLL
      cpeq    gr98, gr97, 0         ; nbytes_to_write == 0?
      jmpt    gr98, $2              ; yes, then done
      nop
      jmp     tx_loop
      const   gr96, SPST
 .endif
```

```
$2:
        const   gr96, firstmsg_flag
        consth  gr96, firstmsg_flag
        load    0, 0, gr96, gr96
        jmpf    gr96, restore_regs
        const   gr96, _msg_sbuf_p
        consth  gr96, _msg_sbuf_p
        const   gr97, 0
        store   0, 0, gr97, gr96      ; clear _msg_sbuf_p for 1st msg
        const   gr96, firstmsg_flag
        consth  gr96, firstmsg_flag
        const   gr97, 0
        store   0, 0, gr97, gr96      ; clear firstmsg_flag
restore_regs:
  .ifdef SERIAL_POLL
        ; clear msg_sbuf_p
        const   gr96, _msg_sbuf_p
        consth  gr96, _msg_sbuf_p
        const   gr97, 0
        store   0, 0, gr97, gr96      ; clear msg_sbuf_p
  .endif
        ; restore gr96–gr98
        const   gr96, scc200_tmp_regs+4
        consth  gr96, scc200_tmp_regs+4
        load    0, 0, gr97, gr96      ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96      ; restore gr98
        const   gr96, scc200_tmp_regs
        consth  gr96, scc200_tmp_regs
        load    0, 0, gr96, gr96      ; restore gr96

        jmpi    lr0
        nop

acknack_code:
        store   0, 0, gr97, gr96      ; write nbytes to send
        add     lr2, lr2, 4
        load    0, 0, gr96, lr2
        const   gr97, ack_flag
        consth  gr97, ack_flag
        jmpt    gr96, set_nack_flag
        constn  gr98, –1
        store   0, 0, gr98, gr97      ; set ack_flag

        jmp     tx_loop
        const   gr96, SPST
```

```
set_nack_flag:
        const   gr97, nack_flag
        consth  gr97, nack_flag
        constn  gr98, –1
        store   0, 0, gr98, gr97      ; set nack_flag

        jmp     tx_loop
        const   gr96, SPST

; ---------------------------------------------------------- MSG_SCC200_TX_INTR
msg_scc200_tx_intr:
        const   gr4, intr_tmp_regs
        consth  gr4, intr_tmp_regs
        store   0, 0, gr96, gr4       ; backup gr96
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        store   0, 0, gr97, gr96      ; backup gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96      ; backup gr98

        const   gr96, ICT
        consth  gr96, ICT
        const   gr97, TXDI
        consth  gr97, TXDI
        store   0, 0, gr97, gr96      ; clear TXDI

        ; check for more bytes to send.
        const   gr96, nbytes_to_write
        consth  gr96, nbytes_to_write
        load    0, 0, gr97, gr96      ; get bytes left
        cpeq    gr98, gr97, 0         ; compare with zero
        jmpt    gr98, $3              ; yes, none left check nack/ack
        nop

        ; get next byte
        sub     gr97, gr97, 1
        store   0, 0, gr97, gr96      ; nbytes_to_write––
        const   gr96, nextchar_p
        consth  gr96, nextchar_p
        load    0, 0, gr97, gr96
        load    0, 1, gr98, gr97      ; get character
        add     gr97, gr97, 1
        store   0, 0, gr97, gr96      ; nextchar_p++
```

scc200.s File continued

```
        ; stuff byte
        const   gr96, SPTH
        consth  gr96, SPTH
        store   0, 0, gr98, gr96        ; put char
$4:
        ; restore gr96–gr98 registers.
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        load    0, 0, gr97, gr96        ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96        ; restore gr98
        const   gr96, intr_tmp_regs
        consth  gr96, intr_tmp_regs
        load    0, 0, gr96, gr96        ; restore gr96

        iret

$3:
        ; check ack_flag if one just sent and clear it.
        const   gr96, ack_flag
        consth  gr96, ack_flag
        load    0, 0, gr97, gr96        ; get flag
        jmpt    gr97, valid_msg         ; set, valid msg intr
        nop
        jmp     $4
        nop

valid_msg:
        ; clear ack_flag
        const   gr97, 0
        store   0, 0, gr97, gr96        ; clear flag

        ; restore gr96–gr98 registers.
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        load    0, 0, gr97, gr96        ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96        ; restore gr98
        const   gr96, intr_tmp_regs
        consth  gr96, intr_tmp_regs
        load    0, 0, gr96, gr96        ; restore gr96

        jmp     msg_V_arrive            ; post interrupt to message
        nop                             ; system
```

scc200.s File continued

```
; ------------------------------------------------------------ MSG_PPI200_INTR
msg_ppi200_intr:
        const   gr4, intr_tmp_regs
        consth  gr4, intr_tmp_regs
        store   0, 0, gr96, gr4         ; backup gr96
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        store   0, 0, gr97, gr96        ; backup gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96        ; backup gr98

        const   gr96, ICT
        consth  gr96, ICT
        const   gr97, PPI
        consth  gr97, PPI
        store   0, 0, gr97, gr96        ; clear PPI

        ; receive the character (FWT=0) and put in buffer.
        const   gr96, PPCT
        consth  gr96, PPCT
        load    0, 0, gr96, gr96        ; read PPCT
        sll     gr97, gr96, (31-7)      ; move FBUSY bit to MSB
        jmpt    gr97, ppi_iret          ; leave character in port
        nop
        const   gr96, PPDT              ; get pdata
        consth  gr96, PPDT
        load    0, 1, gr96, gr96        ; gr96 has received character

        jmp     handle_rx_char
        nop

ppi_iret:
        ; restore register gr96-gr98
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        load    0, 0, gr97, gr96        ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96        ; restore gr98
        const   gr96, intr_tmp_regs
        consth  gr96, intr_tmp_regs
        load    0, 0, gr96, gr96        ; restore gr96

        iret
```

```
; -----------------------------------------------------------MSG_SCC200_RX_INTR
msg_scc200_rx_intr:
        const   gr4, intr_tmp_regs
        consth  gr4, intr_tmp_regs
        store   0, 0, gr96, gr4       ; backup gr96
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        store   0, 0, gr97, gr96      ; backup gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96      ; backup gr98

        const   gr96, ICT
        consth  gr96, ICT
        const   gr97, RXDI
        consth  gr97, RXDI
        store   0, 0, gr97, gr96      ; clear RXDI

        ; receive the character and put in buffer.
        const   gr96, SPRB
        consth  gr96, SPRB
        load    0, 0, gr96, gr96      ; gr96 has received character

handle_rx_char:
        ; put in _msg_next_p location.
        const   gr97, _msg_next_p
        consth  gr97, _msg_next_p
        load    0, 0, gr98, gr97
        store   0, 1, gr96, gr98      ; save character
        add     gr98, gr98, 1         ; update _msg_next_p
        store   0, 0, gr98, gr97

        ; check the buffer for a minimon message.
        const   gr96, _msg_next_p
        consth  gr96, _msg_next_p
        load    0, 0, gr97, gr96      ; msg_next_p
        const   gr96, _msg_rbuf
        consth  gr96, _msg_rbuf       ; msg_rbuf
        sub     gr98, gr97, gr96      ; msg_rbuf-msg_next_p = len

        cplt    gr97, gr98, 8         ; len < 8
        jmpf    gr97, check_for_msg   ; no, check for message
        nop
```

```
do_iret:
        ; restore gr96–gr98 registers
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        load    0, 0, gr97, gr96        ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96        ; restore gr98
        const   gr96, intr_tmp_regs
        consth  gr96, intr_tmp_regs
        load    0, 0, gr96, gr96        ; restore gr96

        iret

check_for_msg:
        ; a message header is in buffer.
        ; gr98 has total length.
        ; gr96 has msg_rbuf
        load    0, 0, gr97, gr96        ; get msg code
        jmpt    gr97, ack_nack_recd     ; handle ack/nack msg
        nop

; -------------------------------------------------------------------------
; message.
        const   gr96, _msg_rbuf+4
        consth  gr96, _msg_rbuf+4
        load    0, 0, gr96, gr96        ; msg length
        add     gr96, gr96, 8+4         ; add msg header size and checksum
        cpgeu   gr97, gr98, gr96        ; have we received all the bytes
        jmpf    gr97, do_iret           ; no return
        nop

        ; compute checksum for message
        const   gr97, intr_tmp_regs+3*4
        consth  gr97, intr_tmp_regs+3*4
        store   0, 0, gr99, gr97        ; backup gr99
        const   gr99, 0                 ; initialize checksum
        sub     gr96, gr96, 4           ; sub checksum size
        const   gr97, _msg_rbuf
        consth  gr97, _msg_rbuf

        sub     gr96, gr96, 2
$6:
        load    0, 1, gr98, gr97
        add     gr99, gr99, gr98
        jmpfdec gr96, $6
        add     gr97, gr97, 1
```

scc200.s File continued

```
; get checksum send by montip
load    0, 1, gr96, gr97
sll     gr96, gr96, 24
add     gr97, gr97, 1
load    0, 1, gr98, gr97
sll     gr98, gr98, 16
or      gr96, gr96, gr98
add     gr97, gr97, 1
load    0, 1, gr98, gr97
sll     gr98, gr98, 8
or      gr96, gr96, gr98
add     gr97, gr97, 1
load    0, 1, gr98, gr97
or      gr96, gr96, gr98

cpeq    gr97, gr96, gr99      ; compare checksums
; reset msg_next_p to beginning of msg_rbuf
const   gr96, _msg_rbuf
consth  gr96, _msg_rbuf
const   gr98, _msg_next_p
consth  gr98, _msg_next_p
jmpt    gr97, ack_it          ; same, valid message
store   0, 0, gr96, gr98      ; reset msg_next_p

; send a nack msg to montip.
; restore gr96–gr99 registers
const   gr96, intr_tmp_regs+4
consth  gr96, intr_tmp_regs+4
load    0, 0, gr97, gr96      ; restore gr97
add     gr96, gr96, 4
load    0, 0, gr98, gr96      ; restore gr98
add     gr96, gr96, 4
load    0, 0, gr99, gr96      ; restore gr99
const   gr96, intr_tmp_regs
consth  gr96, intr_tmp_regs
load    0, 0, gr96, gr96      ; restore gr96

; save lr0, lr2, lr3
const   gr4, intr_tmp_regs
consth  gr4, intr_tmp_regs
store   0, 0, lr0, gr4        ; save lr0
const   lr0, intr_tmp_regs+4
consth  lr0, intr_tmp_regs+4
store   0, 0, lr2, lr0        ; save lr2
add     lr0, lr0, 4
store   0, 0, lr3, lr0        ; save lr3
```

```
        const   lr2, nack_msg_p
        consth  lr2, nack_msg_p
        const   lr3, 8
        call    lr0, msg_scc200_write
        nop

        const   lr0, intr_tmp_regs+4
        consth  lr0, intr_tmp_regs+4
        load    0, 0, lr2, lr0          ; restore lr2
        add     lr0, lr0, 4
        load    0, 0, lr3, lr0          ; restore lr3
        const   lr0, intr_tmp_regs
        consth  lr0, intr_tmp_regs
        load    0, 0, lr0, lr0          ; restore lr0

        iret

ack_it:
        ; restore gr96–gr99 registers
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        load    0, 0, gr97, gr96        ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96        ; restore gr98
        add     gr96, gr96, 4
        load    0, 0, gr99, gr96        ; restore gr99
        const   gr96, intr_tmp_regs
        consth  gr96, intr_tmp_regs
        load    0, 0, gr96, gr96        ; restore gr96

        ; send an ack to montip
        ; save lr0, lr2, lr3
        const   gr4, intr_tmp_regs
        consth  gr4, intr_tmp_regs
        store   0, 0, lr0, gr4          ; save lr0
        const   lr0, intr_tmp_regs+4
        consth  lr0, intr_tmp_regs+4
        store   0, 0, lr2, lr0          ; save lr2
        add     lr0, lr0, 4
        store   0, 0, lr3, lr0          ; save lr3

        const   lr2, ack_flag
        consth  lr2, ack_flag
        constn  lr3, –1
        store   0, 0, lr3, lr2          ; set ack_flag
```

```
        const   lr2, ack_msg_p          ; pointer to ack msg str
        consth  lr2, ack_msg_p
        const   lr3, 8                  ; nbytes in ack msg
        call    lr0, msg_scc200_write ; sends the first character and
        nop                             ; returns

        const   lr0, intr_tmp_regs+4
        consth  lr0, intr_tmp_regs+4
        load    0, 0, lr2, lr0          ; restore lr2
        add     lr0, lr0, 4
        load    0, 0, lr3, lr0          ; restore lr3
        const   lr0, intr_tmp_regs
        consth  lr0, intr_tmp_regs
        load    0, 0, lr0, lr0          ; restore lr0
        iret
; ---------------------------------------------------------------------------
ack_nack_recd:
        const   gr96, _msg_rbuf
        consth  gr96, _msg_rbuf
        const   gr97, _msg_next_p
        consth  gr97, _msg_next_p
        store   0, 0, gr96, gr97        ; initialize msg_next_p

        add     gr96, gr96, 4
        load    0, 0, gr97, gr96        ; get msg len field
        jmpf    gr97, ack_recd          ; ack received
        nop

nack_recd:
        ; restore gr96–gr99 registers
        const   gr96, intr_tmp_regs+4
        consth  gr96, intr_tmp_regs+4
        load    0, 0, gr97, gr96        ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96        ; restore gr98
        const   gr96, intr_tmp_regs
        consth  gr96, intr_tmp_regs
        load    0, 0, gr96, gr96        ; restore gr96
```

```
        ; save lr0, lr2, lr3
        const   gr4, intr_tmp_regs
        consth  gr4, intr_tmp_regs
        store   0, 0, lr0, gr4        ; save lr0
        const   lr0, intr_tmp_regs+4
        consth  lr0, intr_tmp_regs+4
        store   0, 0, lr2, lr0        ; save lr2
        add     lr0, lr0, 4
        store   0, 0, lr3, lr0        ; save lr3

        const   lr2, _msg_lastsent_p  ; address of msg
        consth  lr2, _msg_lastsent_p
        load    0, 0, lr2, lr2
        add     lr3, lr2, 4
        load    0, 0, lr3, lr3        ; msg length
        add     lr3, lr3, 8           ; msglen+msg header
        call    lr0, msg_scc200_write
        nop

        const   lr0, intr_tmp_regs+4
        consth  lr0, intr_tmp_regs+4
        load    0, 0, lr2, lr0        ; restore lr2
        add     lr0, lr0, 4
        load    0, 0, lr3, lr0        ; restore lr3
        const   lr0, intr_tmp_regs
        consth  lr0, intr_tmp_regs
        load    0, 0, lr0, lr0        ; restore lr0

        iret

ack_recd:
        ; clear _msg_sbuf_p semaphore
        const   gr96, _msg_sbuf_p
        consth  gr96, _msg_sbuf_p
        const   gr97, 0
        store   0, 0, gr97, gr96      ; clear semaphore

        jmp     do_iret
        nop
```

```
; ----------------------------------------------------- MSG_SCC200_WAIT_FOR
; In interrupt mode, returns immediately.
; In polled mode, blocks until a msg is received.
; returns gr96 = 0 if no message, –1 if valid message in buffer
msg_scc200_wait_for:
  .ifndef SERIAL_POLL
        ; simpler case – interrupt mode
        jmpi    lr0
        const   gr96, 0               ; no message
  .else
        ; block until a message is received – polled mode.
        const   gr96, poll_tmp_glob
        consth  gr96, poll_tmp_glob
        store   0, 0, gr97, gr96      ; backup gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96      ; backup gr98
        add     gr96, gr96, 4
        store   0, 0, gr99, gr96      ; backup gr99

poll_loop:
        ; poll for a character
        const   gr96, SPST
$7:
        consth  gr96, SPST
        load    0, 0, gr96, gr96      ; read SPST
        sll     gr96, gr96, RDRShift  ; rdr bit
        jmpf    gr96, $7
        const   gr96, SPST

        ; from here on the code is very similar to the rx_intr code above.
        ; except that you don’t iret for one thing, and you wait until
        ; a message is received – not an ack or nack, but a message
        ; for the debug core to process.

        ; character found in receive buffer
        ; receive the character and put in buffer.
        const   gr96, SPRB
        consth  gr96, SPRB
        load    0, 0, gr96, gr96      ; gr96 has received character

        ; put in _msg_next_p location.
        const   gr97, _msg_next_p
        consth  gr97, _msg_next_p
        load    0, 0, gr98, gr97
        store   0, 1, gr96, gr98      ; save character
        add     gr98, gr98, 1         ; update _msg_next_p
        store   0, 0, gr98, gr97
```

```
        ; check the buffer for a minimon message.
        const   gr96, _msg_next_p
        consth  gr96, _msg_next_p
        load    0, 0, gr97, gr96        ; msg_next_p
        const   gr96, _msg_rbuf
        consth  gr96, _msg_rbuf         ; msg_rbuf
        sub     gr98, gr97, gr96        ; msg_rbuf-msg_next_p = len

        cplt    gr97, gr98, 8           ; len < 8
        jmpf    gr97, poll_check_for_msg      ; no, check for message
        nop

        ; Not enough characters received, continue polling.
continue_poll:
        jmp     poll_loop
        nop

poll_check_for_msg:
        ; a message header is in buffer.
        ; gr98 has total length.
        ; gr96 has msg_rbuf
        load    0, 0, gr97, gr96        ; get msg code
        jmpt    gr97, poll_ack_nack_recd      ; handle ack/nack msg
        nop

; -----------------------------------------------------------------------
; message.
        const   gr96, _msg_rbuf+4
        consth  gr96, _msg_rbuf+4
        load    0, 0, gr96, gr96        ; msg length
        add     gr96, gr96, 8+4         ; add msg header size and checksum
        cpgeu   gr97, gr98, gr96        ; have we received all the bytes

        jmpf    gr97, poll_loop         ; no continue polling
        nop

        ; compute checksum for message
        const   gr99, 0                 ; initialize checksum
        sub     gr96, gr96, 4           ; sub checksum size
        const   gr97, _msg_rbuf
        consth  gr97, _msg_rbuf
```

```
        sub     gr96, gr96, 2
$8:
        load    0, 1, gr98, gr97
        add     gr99, gr99, gr98
        jmpfdec gr96, $8
        add     gr97, gr97, 1

        ; get checksum send by montip
        load    0, 1, gr96, gr97
        sll     gr96, gr96, 24
        add     gr97, gr97, 1
        load    0, 1, gr98, gr97
        sll     gr98, gr98, 16
        or      gr96, gr96, gr98
        add     gr97, gr97, 1
        load    0, 1, gr98, gr97
        sll     gr98, gr98, 8
        or      gr96, gr96, gr98
        add     gr97, gr97, 1
        load    0, 1, gr98, gr97
        or      gr96, gr96, gr98

        cpeq    gr97, gr96, gr99       ; compare checksums
        ; reset msg_next_p to beginning of msg_rbuf
        const   gr96, _msg_rbuf
        consth  gr96, _msg_rbuf
        const   gr98, _msg_next_p
        consth  gr98, _msg_next_p
        jmpt    gr97, poll_ack_it      ; same, valid message
        store   0, 0, gr96, gr98       ; reset msg_next_p

        ; send a nack msg to montip.
        ; save lr0, lr2, lr3
        const   gr96, poll_tmp_loc
        consth  gr96, poll_tmp_loc
        store   0, 0, lr0, gr96        ; save lr0
        add     gr96, gr96, 4
        store   0, 0, lr2, gr96        ; save lr2
        add     gr96, gr96, 4
        store   0, 0, lr3, gr96        ; save lr3

        const   lr2, nack_msg_p
        consth  lr2, nack_msg_p
        const   lr3, 8
        call    lr0, msg_scc200_write ; poll mode write
        nop
```

scc200.s File continued

```
        const   gr96, poll_tmp_loc
        consth  gr96, poll_tmp_loc
        load    0, 0, lr0, gr96        ; restore lr0
        add     gr96, gr96, 4
        load    0, 0, lr2, gr96        ; restore lr2
        add     gr96, gr96, 4
        load    0, 0, lr3, gr96        ; restore lr3

        ; continue polling for a valid message
        jmp     poll_loop
        nop

poll_ack_it:
        ; restore gr97–gr99 registers
        const   gr96, poll_tmp_glob
        consth  gr96, poll_tmp_glob
        load    0, 0, gr97, gr96       ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96       ; restore gr98
        add     gr96, gr96, 4
        load    0, 0, gr99, gr96       ; restore gr99

        ; send an ack to montip
        ; save lr0, lr2, lr3
        const   gr96, poll_tmp_loc
        consth  gr96, poll_tmp_loc
        store   0, 0, lr0, gr96        ; save lr0
        add     gr96, gr96, 4
        store   0, 0, lr2, gr96        ; save lr2
        add     gr96, gr96, 4
        store   0, 0, lr3, gr96        ; save lr3

        const   lr2, ack_flag
        consth  lr2, ack_flag
        constn  lr3, –1
        store   0, 0, lr3, lr2         ; set ack_flag

        const   lr2, ack_msg_p         ; pointer to ack msg str
        consth  lr2, ack_msg_p
        const   lr3, 8                 ; nbytes in ack msg
        call    lr0, msg_scc200_write  ; polled mode write
        nop
```

```
        const   gr96, poll_tmp_loc
        consth  gr96, poll_tmp_loc
        load    0, 0, lr0, gr96        ; restore lr0
        add     gr96, gr96, 4
        load    0, 0, lr2, gr96        ; restore lr2
        add     gr96, gr96, 4
        load    0, 0, lr3, gr96        ; restore lr3

        jmpi    lr0                    ; RETURN WITH A VALID MSG
        constn  gr96, –1               ; TRUE
; ---------------------------------------------------------------------------
poll_ack_nack_recd:
        const   gr96, _msg_rbuf
        consth  gr96, _msg_rbuf
        const   gr97, _msg_next_p
        consth  gr97, _msg_next_p
        store   0, 0, gr96, gr97       ; initialize msg_next_p

        add     gr96, gr96, 4
        load    0, 0, gr97, gr96       ; get msg len field
        jmpf    gr97, poll_ack_recd    ; ack received
        nop

poll_nack_recd:
        ; save lr0, lr2, lr3
        ; save lr0, lr2, lr3
        const   gr96, poll_tmp_loc
        consth  gr96, poll_tmp_loc
        store   0, 0, lr0, gr96        ; save lr0
        add     gr96, gr96, 4
        store   0, 0, lr2, gr96        ; save lr2
        add     gr96, gr96, 4
        store   0, 0, lr3, gr96        ; save lr3

        const   lr2, _msg_lastsent_p   ; address of msg
        consth  lr2, _msg_lastsent_p
        load    0, 0, lr2, lr2
        add     lr3, lr2, 4
        load    0, 0, lr3, lr3         ; msg length
        add     lr3, lr3, 8            ; msglen+msg header
        call    lr0, msg_scc200_write  ; polled write
        nop
```

```
        const   gr96, poll_tmp_loc
        consth  gr96, poll_tmp_loc
        load    0, 0, lr0, gr96        ; restore lr0
        add     gr96, gr96, 4
        load    0, 0, lr2, gr96        ; restore lr2
        add     gr96, gr96, 4
        load    0, 0, lr3, gr96        ; restore lr3

        jmp     poll_loop
        nop

poll_ack_recd:
        ; clear _msg_sbuf_p semaphore
        const   gr96, _msg_sbuf_p
        consth  gr96, _msg_sbuf_p
        const   gr97, 0
        store   0, 0, gr97, gr96       ; clear semaphore

        jmp     poll_loop              ; continue_polling
        nop

  .endif
```

sa200hw.s File

```
        .ident  ”@(#)sa200hw.s 1.5 93/08/18 09:21:05, Srini, AMD”

        .file   ”sa200hw.s”

        .include ”stats.ah”

        .equ    COMM_VERSION, 0x06

        ; offsets into the intr3 vector table using CLZ
        .equ    TXDI_OFFSET,   (31–5)*4
        .equ    RXDI_OFFSET,   (31–6)*4
        .equ    RXSI_OFFSET,   (31–7)*4
        .equ    PPI_OFFSET,    (31–11)*4

        .extern msg_scc200_init
        .extern msg_scc200_write
        .extern msg_scc200_wait_for
        .extern msg_scc200_tx_intr
        .extern msg_scc200_rx_intr
        .extern msg_ppi200_intr
        .extern msg_lpt200_init

        .extern dbg_trap

        .global msg_initcomm   ; initialize comm interface.
        .global serial_int     ; serial interface interrupt handler.

        .global msg_write_p
        .global msg_wait_for_p

        .bss
msg_write_p:
        .block  1*4
msg_wait_for_p:
        .block  1*4
intr3_V_table:
        .block  32*4           ; hold 32 interrupt vectors (max)
save_regs:
        .block  3*4
```

sa200hw.s File continued

```
        .text
; ---------------------------------------------------------- MSG_INITCOMM
; return version in gr96.
msg_initcomm:
        const   gr96, save_regs
        consth  gr96, save_regs
        store   0, 0, gr97, gr96        ; backup gr97
        add     gr96, gr96, 4
        store   0, 0, gr98, gr96        ; backup gr98
        add     gr96, gr96, 4
        store   0, 0, lr0, gr96         ; backup lr0

        ; initialize the msg_write_p with write functions.
        const   gr96, msg_write_p
        consth  gr96, msg_write_p
        const   gr97, msg_scc200_write
        consth  gr97, msg_scc200_write
        store   0, 0, gr97, gr96        ; only one for now

        ; initialize msg_wait_for_p pointer
        const   gr96, msg_wait_for_p
        consth  gr96, msg_wait_for_p
        const   gr97, msg_scc200_wait_for
        consth  gr97, msg_scc200_wait_for
        store   0, 0, gr97, gr96

        ; initialize table with default entries.
        const   gr96, intr3_V_table
        consth  gr96, intr3_V_table
        const   gr97, default_intr3
        consth  gr97, default_intr3
        const   gr98, 32–2
$1:
        store   0, 0, gr97, gr96
        jmpfdec gr98, $1
        add     gr96, gr96, 4

        ; install known handlers.
        const   gr96, intr3_V_table+TXDI_OFFSET
        consth  gr96, intr3_V_table+TXDI_OFFSET
        const   gr97, msg_scc200_tx_intr
        consth  gr97, msg_scc200_tx_intr
        store   0, 0, gr97, gr96        ; tx intr
```

```
        const   gr96, intr3_V_table+RXDI_OFFSET
        consth  gr96, intr3_V_table+RXDI_OFFSET
        const   gr97, msg_scc200_rx_intr
        consth  gr97, msg_scc200_rx_intr
        store   0, 0, gr97, gr96      ; rx intr

        const   gr96, intr3_V_table+PPI_OFFSET
        consth  gr96, intr3_V_table+PPI_OFFSET
        const   gr97, msg_ppi200_intr
        consth  gr97, msg_ppi200_intr
        store   0, 0, gr97, gr96      ; ppi intr

        ; initialize the peripherals.
        const   gr96, msg_scc200_init
        consth  gr96, msg_scc200_init
        calli   lr0, gr96
        nop

        ; initialize 29200 parallel port
        const   gr96, msg_lpt200_init
        consth  gr96, msg_lpt200_init
        calli   lr0, gr96
        nop

        ; restore registers
        const   gr96, save_regs
        consth  gr96, save_regs
        load    0, 0, gr97, gr96      ; restore gr97
        add     gr96, gr96, 4
        load    0, 0, gr98, gr96      ; restore gr98
        add     gr96, gr96, 4
        load    0, 0, lr0, gr96       ; restore lr0

        jmpi    lr0
        const   gr96, COMM_VERSION    ; return version number

        .bss
intr_save:      .block  4*4
```

```
        .text
; ---------------------------------------------------------- SERIAL_INT
serial_int:
; We use count of leading zeroes to determine the offset in the interrupt
; table, and branch to the interrupt handler.
        const   gr4, intr_save
        consth  gr4, intr_save
        store   0, 0, gr96, gr4         ; backup gr96
        const   gr96, intr_save+4
        consth  gr96, intr_save+4
        store   0, 0, gr97, gr96        ; backup gr97

        const   gr96, ICT
        consth  gr96, ICT
        load    0, 0, gr96, gr96        ; read ICT
        clz     gr96, gr96
        cpeq    gr97, gr96, 32
        jmpt    gr97, $2                ; no interrupts??
        nop
        sll     gr96, gr96, 2           ; find offset into table
        const   gr97, intr3_V_table
        consth  gr97, intr3_V_table
        add     gr97, gr97, gr96        ; handler address pointer

        const   gr96, intr_save
        consth  gr96, intr_save
        load    0, 0, gr96, gr96        ; restore gr96

        load    0, 0, gr4, gr97         ; address

        const   gr97, intr_save+4
        consth  gr97, intr_save+4
        load    0, 0, gr97, gr97        ; restore gr97

        jmpi    gr4
        nop
$2:
        ; restore regs
        const   gr96, intr_save+4
        consth  gr96, intr_save+4
        load    0, 0, gr97, gr96        ; restore gr97
        const   gr96, intr_save
        consth  gr96, intr_save
        load    0, 0, gr96, gr96        ; restore gr96
        iret
```

sa200hw.s File continued

```
default_intr3:
; clear the interrupt and call dbg_trap
        const   gr96, ICT
        consth  gr96, ICT
        load    0, 0, gr96, gr96      ; read ICT
        clz     gr96, gr96
        cpeq    gr97, gr97, gr97      ; sets most significant bit
        srl     gr97, gr97, gr96      ; set bit to reset
        const   gr96, ICT
        consth  gr96, ICT
        store   0, 0, gr97, gr96

        ; restore regs
        const   gr96, intr_save+4
        consth  gr96, intr_save+4
        load    0, 0, gr97, gr96      ; restore gr97
        const   gr96, intr_save
        consth  gr96, intr_save
        load    0, 0, gr96, gr96      ; restore gr96

        iret    ; simply iret for now.
```

Index

Symbols

A

B

C

D

L

M

N

O

P

R

S

T

U

V

W

X

Y