Howto: GDB Remote Serial Protocol

Writing a RSP Server

Jeremy Bennett
Embecosm

Application Note 4. Issue 2
Published November 2008

Legal Notice

Table of Contents

List of Figures

EMBECOSM

Chapter 1. Introduction

This document complements the existing documentation for GDB ([3], [4]). It is intended to help software engineers implementing a server for the GDB Remote Serial Protocol (RSP) for the first time.

This application note is based on the author's experience to date. It will be updated in future issues. Suggestions for improvements are always welcome.

1.1. Rationale

The GDB User Guide [3] documents the Remote Serial Protocol (RSP) for communicating with remote targets. The target must act as a server for the RSP, and the source distribution includes stub implementations for architectures such as the Motorola 680xx and Sun SPARC. The User Guide offers advice on how these stubs can be modified and integrated for new targets.

However the examples have not been changed for several years, and the advice on using the stubs is now out of date. The documentation also lacks any explanation of the dynamics of the protocol—the sequence of commands/responses used to effect the various GDB commands.

This document aims to fill that gap, by explaining how the RSP works today and how it can be used to write a server for a target to be debugged with GDB.

Throughout, examples are provided from the author's experience implementing a RSP server for the OpenRISC 1000 architecture. This document captures the learning experience, with the intention of helping others.

1.2. Target Audience

If you are about to start a port of GDB to a new architecture, this document is for you. If at the end of your endeavors you are better informed, please help by adding to this document.

If you have already been through the porting process, please help others by adding to this document.

1.3. Further Sources of Information

1.3.1. Written Documentation

The main user guide for GDB [3] explains how remote debugging works and provides the reference for the various RSP packets.

The main GDB code base is generally well commented, particularly in the headers for the major interfaces. Inevitably this must be the definitive place to find out exactly how a particular function behaves. In particular the source code for the RSP client side in **`gdb/remote.c`** provides the definitive guide on the expected dynamics of the protocol.

The files making up the RSP server for the OpenRISC 1000 are comprehensively commented, and can be processed with Doxygen [5]. Each function's behavior, its parameters and any return value is described.

This application note complements the Embecosm Application Note 3, "HOWTO: Porting the GNU Debugger" [2]. Details of the OpenRISC 1000 can be found in its Architecture Manual [6]. The OpenRISC 1000 architectural simulator and tool chain is documented in Embecosm Application Note 2 [1].

1.3.2. Other Information Channels

The main GDB website is at sourceware.org/gdb/. It is supplemented by the less formal GDB Wiki at sourceware.org/gdb/wiki/.

The GDB developer community communicate through the GDB mailing lists and using IRC chat. These are always good places to find solutions to problems.

IRC is channel **`#gdb`** on **`irc.freenode.net`**.

The main mailing list for discussion is gdb@sourceware.org, although for detailed insight, take a look at the patches mailing list, gdb-patches@sourceware.org. See the main GDB website for details of subscribing to these mailing lists.

1.4. About Embecosm

Embecosm is a consultancy specializing in open source tools, models and training for the embedded software community. All Embecosm products are freely available under open source licenses.

Embecosm offers a range of commercial services.

- Customization of open source tools and software, including porting to new architectures.
- Support, tutorials and training for open source tools and software.
- Custom software development for the embedded market, including bespoke software models of hardware.
- Independent evaluation of software tools.

For further information, visit the Embecosm website at www.embecosm.com.

Chapter 2. Overview of the Remote Serial Protocol

The GDB Remote Serial Protocol (RSP) provides a high level protocol allowing GDB to connect to any target remotely. If a target's architecture is defined in GDB and the target implements the server side of the RSP protocol, then the debugger will be able to connect remotely to that target.

The protocol supports a wide range of connection types: direct serial devices, UDP/IP, TCP/IP and POSIX pipes. Historically RSP has only required 7-bit clean connections. However more recent commands added to the protocol assume an 8-bit clean connection. It is also worth noting, that although UDP/IP is supported, lost packets with unreliable transport methods such as this may lead to GDB reporting errors.

RSP is most commonly of value in embedded environments, where it is not possible to run GDB natively on the target.

The protocol is layered, approximately following the OSI model as shown in Figure 2.1.

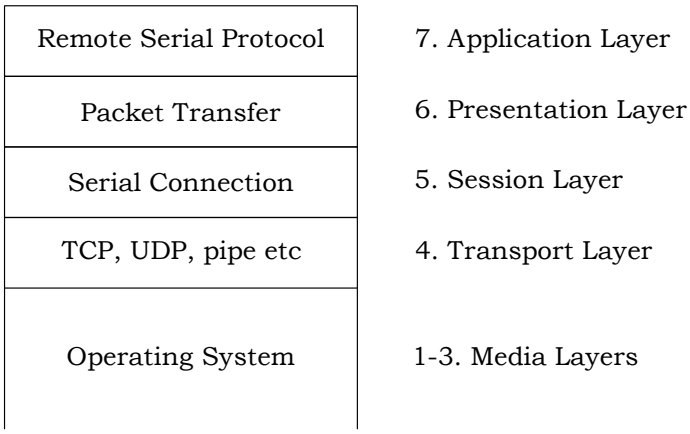

Figure 2.1. OSI Layers in the Remote Serial Protocol

2.1. Client-Server Relationship

The GDB program acts as the RSP client with the target acting as the RSP server. The client issues packets which are requests for information or action. Depending on the nature of the client packet, the server may respond with a packet of its own.

This is the only circumstance under which the server sends a packet: in reply to a packet from the client requiring a response.

2.2. Session Layer: The Serial Connection

The serial connection is established in response to a **target remote** or **target extended-remote** command from the GDB client. The way the server handles this depends on the nature of the serial connection:

- *Connection via a serial device.* The target should be listening for connections on the device. This may either be via routine polling or via an event driven interface. Once the connection is established, packets are read from and written to the device.

- *Connection via TCP/IP or UDP/IP*. The target should be listening on a socket connected to the specified port. This may either be via routine polling or via an event driven interface. Accepting a new connection (the POSIX **`accept ()`** function) will yield a file descriptor, which can be used for reading and writing packets.
- *Connection via a pipe*. The target will be created, with standard input and output as the file descriptors for packet reading and writing.

In each case there is no specific requirement that the target be either running or stopped. GDB will establish via RSP commands the state of the target once the connection is established.

GDB is almost entirely non-preemptive, which is reflected in the sequence of packet exchanges of RSP. The exception is when GDB wishes to interrupt an executing program (typically via ctrl-C). A single byte, 0x03, is sent (no packet structure). If the target is prepared to handle such interrupts it should recognize such bytes. Unless the target is routinely polling for input (which may be the case for simulators), a prompt response typically will require an event driven reader for the connection.

2.3. Presentation Layer: Packet Transfer

The basic format of a RSP packet is shown in Figure 2.2.

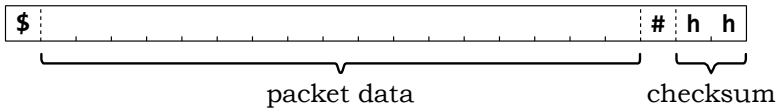

Figure 2.2. RSP Packet Format

For almost all packets, binary data is represented as two hexadecimal digits per byte of data. The checksum is the unsigned sum of all the characters in the packet data modulo 256. It is represented as a pair of hexadecimal digits.

Where the characters '#' or '$' appear in the packet data, they must be escaped. The escape character is ASCII 0x7d ('}'), and is followed by the original character XORed with 0x20. The character '}' itself must also be escaped.

The small number of packets which transmit data as raw binary (thus requiring an 8-bit clean connection) must also escape the characters '#', '$' and '}' if they occur in the binary data.

Reply packets sent by the server may use run-length encoding. The format is to follow the character being repeated by '*' and then the character whose ASCII code is 28 greater than the total repeat, so long as it remains a printable ASCII character (i.e. not greater than 126). Thus the string **`"XXXXX"`** would be represented as **`"X*!"`** (**`'!'`** is ASCII 33).

This feature is suitable for run-lengths of 4, 5 and 8-97. Run lengths of 6 and 7 cannot be used, since the repeat characters would be '#' and '$' and interfere with the recognition of the packet itself before decoding. For these cases, a run length of 5 is used, followed by 1 or 2 instances of the repeated character as required. '*' and '}' cause no problem, since they are part of decoding, and their use in a run-length would be recognized as such.

Note
There is no requirement for a server to use run length encoding.

2.3.1. Packet Acknowledgment

Each packet should be acknowledged with a single character. '+' to indicate satisfactory receipt, with valid checksum or '-' to indicate failure and request retransmission.

Retransmission should be requested until a satisfactory packet is received.

2.3.2. Interrupt

The GDB client may wish to interrupt the server (e.g. when the user has pressed ctrl-C). This is indicated by transmitting the character 0x03 between packets.

If the server wishes to handle such interrupts, it should recognize such characters and process as appropriate. However not all servers are capable of handling such requests. The server is free to ignore such out-of-band characters.

2.4. Application Layer: Remote Serial Protocol

RSP commands from the client to the server are textual strings, optionally followed by arguments. Each command is sent in its own packet. The packets fall into four groups:

1. *Packets requiring no acknowledgment.* These commands are: **f**, **i**, **I**, **k**, **R**, **t** and **vFlashDone**.
2. *Packets requiring a simple acknowledgment packet.* The acknowledgment is either **OK**, **Enn** (where **nn** is an error number) or for some commands an empty packet (meaning "unsupported"). These commands are: **!**, **A**, **D**, **G**, **H**, **M**, **P**, **Qxxxx**, **T**, **vFlashErase**, **vFlashWrite**, **X**, **z** and **Z**.
3. *Packets that return result data or an error code.*. These commands are: **?**, **c**, **C**, **g**, **m**, **p**, **qxxxx**, **s**, **S** and most **vxxxx**.
4. *Deprecated packets which should no longer be used.* These commands are **b**, **B**, **d** and **r**.

This application note does not document all these commands, except where clarification is needed, since thy are all documented in Appendix D of the main GDB user guide ([3]).

Tip

Many commands come in pairs: for example **g** and **G**. In general the lower case is used for the command to read or delete data, or the command in its simpler form. The upper case is used to write or install data, or for a more complex form of the command.

The RSP was developed over several years, and represents an evolved standard, but one which had to keep backward compatibility. As a consequence the detailed syntax can be inconsistent. For example most commands are separated from their arguments by ':', but some use ',' for this purpose.

2.5. Putting it All Together to Build a Server

There are three approaches to adding a RSP server to a target.

1. Run the **gdbserver** program on the target. A variant of this uses a custom server program to drive a physical interface to real hardware. This is most commonly seen with programs, running on the host, which drive a JTAG link connected via a parallel port or USB.
2. Implement code on the target to establish a connection, recognize the packets and implement the behavior.
3. For simulators, add code to the simulator to establish a connection, recognize the packets and implement the behavior in the simulator.

When remote debugging, GDB assumes that the target server will terminate the connection if the target program exits. However there is a variant, invoked by **target extended-remote**, which makes the server persistent, allowing the user to restart a program, or run an alternative program. This is discussed in more detail later (see Section 3.1.2).

In general GDB assumes that when it connects to a target via RSP, that target will be stopped. However there are new features in GDB allowing it to work asynchronously while some or all threads in the target continue executing. This is discussed in more detail later (see Section 3.1.3).

2.5.1. Using gdbserver

The **gdbserver** command is well documented in the GDB User Guide [3]. This approach is suitable for powerful targets, where it is easy to invoke a program from the command line.

Generally this approach is not suitable for embedded systems.

2.5.2. Implementing Server Code on the Target

This is the usual approach for embedded systems, and is the strategy encapsulated in the server stubs supplied with the GDB source code.

There are two key components of the code:

1. Code to establish the serial connection with the client GDB session.
2. Code for the target's interrupt handlers, so all exceptions are routed through the RSP server.

In the stub code, the user must implement the serial connection by supplying functions **`getDebugChar ()`** and **`putDebugChar ()`**. The user must supply the function **`exceptionHandler ()`** to set up exception handling.

The serial connection is usually established on the first call to **`getDebugChar ()`**. This is standard POSIX code to access either the serial device, or to listen for a TCP/IP or UDP/IP connection. The target may choose to block here, if it does not wish to run without control from a GDB client.

If the serial connection chooses not to block on **`getDebugChar ()`** then the exception handler should be prepared for this response, allowing the exception to be processed as normal.

Note

The stub RSP server code supplied with the GDB source distribution assumes **`getDebugChar ()`** blocks until the connection is established.

In general the server interacts with the client only when it has received control due to a target exception.

At start up, the first time this occurs, the target will be waiting for the GDB client to send a packet to which it can respond. These dialogs will continue until the client GDB session wishes to **continue** or **step** the target (**c**, **C**, **i**, **I**, **s** or **S** packet).

Thereafter control is received only when another exception has occurred, following a **continue** or **step**. In this case, the first action of the target RSP server should be to send the reply packet back to the client GDB session.

Caution

The key limitation in the stub RSP server code supplied with the GDB source distribution is that it only deals with the second case. In other words, it always sends a reply packet to the client, even on first execution.

This causes two problems. First, the **`putDebugChar ()`** is called before **`getDebugChar ()`**, so it must be able to establish the connection.

Secondly, the initial reply is sent without a request packet from the client GDB session. As a result this reply will typically be queued and appear as the reply to

the first request packet from GDB. The client interface is quite robust and usually quietly rejects unexpected packets, but there is potential for client requests and server responses to get out of step. It certainly does not represent good program design.

The final issue that server code needs to address is the issue of BREAK signaling from the client. This is a raw 0x03 byte sent from the client between packets. Typically this is in response to a ctrl-C from the client GDB session.

If the target server wishes to handle such signaling, it must provide an event driven **`getDebugChar ()`**, triggered when data is received, which can act on such BREAK signals.

2.5.3. Implementing Server Code for Simulators

Simulators are commonly integrated separately into GDB, and accessed using the **target sim** command.

However it can also be useful to connect to them by using the RSP. This allows the GDB experience to be identical whether simulator or real silicon is used.

The general approach is the same as that for implementing code on a target (see Section 2.5.2). However the code forms part of the simulator, not the target. The RSP handler will be attached to the simulators handling of events, rather than the events themselves.

In general the simulator will use the same form of connection as when debugging real silicon. Where the RSP server for real silicon is implemented on the target, or **gdbserver** is used, connection via a serial device, TCP/IP or UDP/IP is appropriate. Where the RSP interface for real silicon is via a pipe to a program driving JTAG a pipe interface should be used to launch the simulator.

The example used in Chapter 4 is based on a simulator for the OpenRISC 1000.

2.5.4. Implementing a Custom Server for JTAG

Many embedded systems will offer JTAG ports for debugging. Most commonly these are connected to a host workstation running GDB via the parallel port or USB.

In the past users would implement a custom target interface in GDB to drive the JTAG interface directly. However with RSP it makes more sense to write a RSP server program, which runs standalone on the host. This program maps RSP commands and responses to the underlying JTAG interface.

Logically this is rather like a custom **gdbserver**, although it runs on the host rather than the target. The implementation techniques are similar to those required for interfacing to a simulator.

This is one situation, where using the pipe interface is sensible. The pipe interface is used to launch the program which will talk to the JTAG interface. If this approach is used, then debugging via a simulator should also use a pipe interface to launch the simulator, thus allowing the debugging experience to be the same whether real silicon or a simulator is used.

Chapter 3. Mapping GDB Commands to RSP

3.1. Remote Debugging in GDB

GDB provides two flavors of remote debugging via the RSP

1. **target remote**. This is the GDB command documented in the GDB User Guide ([3]).
2. **target extended-remote**. The RSP server is made persistent. When the target exits, the server does not close the connection. The user is able to restart the target program, or load and run an alternative program.

3.1.1. Standard Remote Debugging

A RSP server supporting standard remote debugging (i.e. using the GDB **target remote** command) should implement at least the following RSP packets:

- **?**. Report why the target halted.
- **c**, **C**, **s** and **S**. Continue or step the target (possibly with a particular signal). A minimal implementation may not support stepping or continuing with a signal.
- **D**. Detach from the client.
- **g** and **G**. Read or write general registers.
- **qC** and **H**. Report the current thread or set the thread for subsequent operations. The significance of this will depend on whether the target supports threads.
- **k**. Kill the target. The semantics of this are not clearly defined. Most targets should probably ignore it.
- **m** and **M**. Read or write main memory.
- **p** and **P**. Read or write a specific register.
- **qOffsets**. Report the offsets to use when relocating downloaded code.
- **qSupported**. Report the features supported by the RSP server. As a minimum, just the packet size can be reported.
- **qSymbol::** (i.e. the **qSymbol** packet with no arguments). Request any symbol table data. A minimal implementation should request no data.
- **vCont?**. Report what **vCont** actions are supported. A minimal implementation should return an empty packet to indicate no actions are supported.
- **X**. Load binary data.
- **z** and **Z**. Clear or set breakpoints or watchpoints.

3.1.2. Extended Remote Debugging

A RSP server supporting standard remote debugging (i.e. using the GDB **target remote** command) should implement at least the following RSP packets in addition to those required for standard remote debugging:

- **!**. Advise the target that extended remote debugging is being used.
- **R**. Restart the program being run.
- **vAttach**. Attach to a new process with a specified process ID. This packet need not be implemented if the target has no concept of a process ID, but should return an error code.
- **vRun**. Specify a new program and arguments to run. A minimal implementation may restrict this to the case where only the current program may be run again.

3.1.3. Asynchronous Remote Debugging

The most recent versions of GDB have started to introduce the concept of asynchronous debugging. This is primarily for use with targets capable of "non-stop" execution. Such targets are able to stop the execution of a single thread in a multithreaded environment, allowing it to be debugged while others continue to execute.

This still represents technology under development. In GDB 6.8, the commands **target async** and **target extended-async** were provided to specify remote debugging of a non-stop target in asynchronous fashion.

The mechanism will change in the future, with GDB flags set to specify asynchronous interpretation of commands, which are otherwise unchanged. Readers particularly interested in this area should look at the current development version of GDB and the discussions in the various GDB newsgroups.

Asynchronous debugging requires that the target support packets specifying execution of particular threads. The most significant of these are:

- **H**. To specify which thread a subsequent command should apply to.
- **q** (various packets). The query packets related to threads, **qC**, **qfThreadInfo**, **qsThreadInfo**, **qGetTLSAddr** and **qThreadExtraInfo** will need to be implemented.
- **T**. To report if a particular thread is alive.
- **vCont**. To specify step or continue actions specific to one or more threads.

In addition, non-stop targets should also support the **T** response to **continue** or **step** commands, so that status of individual threads can be reported.

3.2. GDB Standard Remote Command Dialogs

The following sections show diagrammatically how various GDB commands map onto RSP packet exchanges. These implement the desired behavior with standard remote debugging (i.e when connecting with **target remote**).

3.2.1. The target remote Command

The RSP packet exchanges to implement the GDB **target remote** command are shown as a sequence diagram in Figure 3.1.

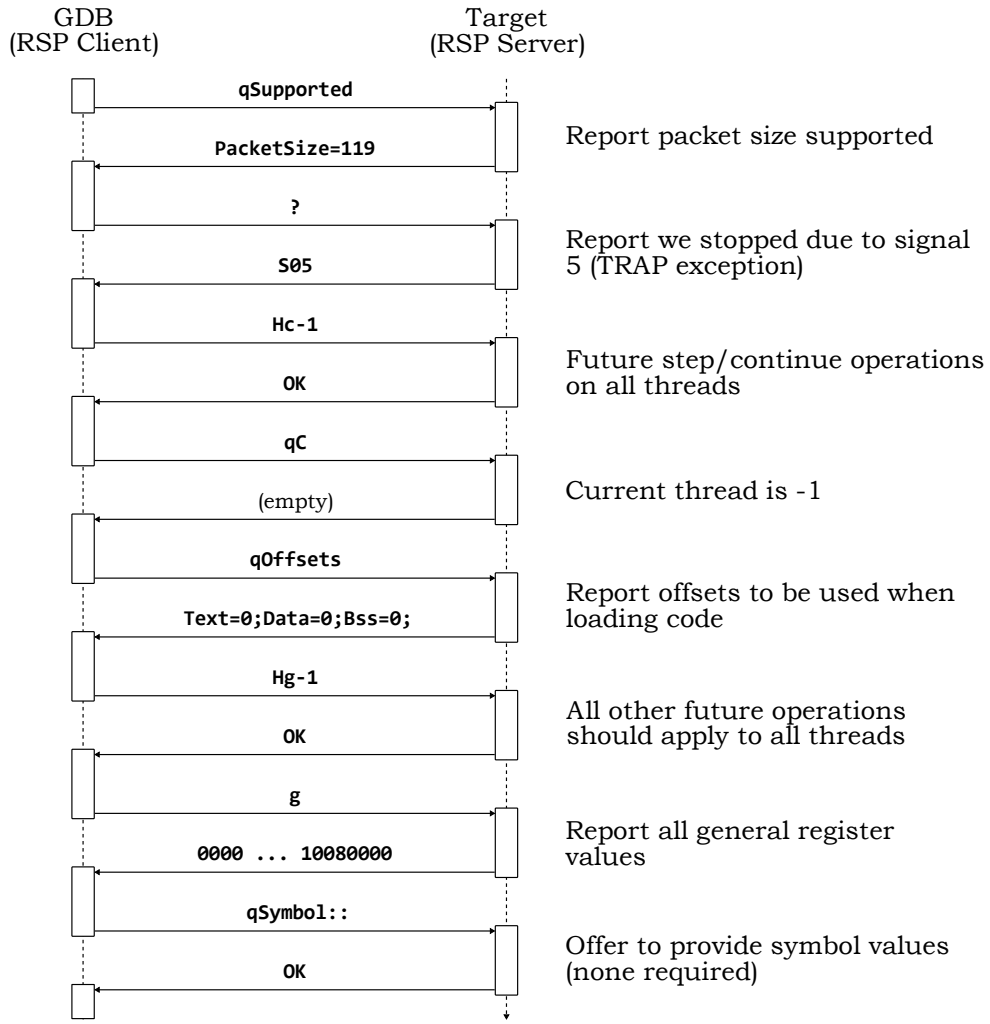

Figure 3.1. RSP packet exchanges for the GDB target remote command

This is the initial dialog once the connection has been established. The first thing the client needs to know is what this RSP server supports. The only feature that matters is to report the packet size that is supported. The largest packet that will be needed is to hold a command with the hexadecimal values of all the general registers (for the **G** packets). In this example, there are a total of 35 32-bit registers, each requiring 8 hex characters + 1 character for the 'G', a total of 281 (hexadecimal 0x119) characters.

The client then asks why the target halted. For a standard remote connection (rather than extended remote connection), the target must be running, even if it has halted for a signal. So the client will verify that the reply is not **W** (exited) or **X** (terminated with signal). In this case the target reports it has stopped due to a TRAP exception.

The next packet is an instruction from the client that any future **step** or **continue** commands should apply to all threads. This is followed by a request (**qC**) for information on the thread currently running. In this example the target is "bare metal", so there is no concept of threads. An empty response is interpreted as "use the existing value", which suits in this case—since it is never set explicitly, it will be the NULL thread ID, which is appropriate.

The next packet (**qOffsets**) requests any offsets for loading binary data. At the minimum this must return offsets for the text, data and BSS sections of an executable—in this example all zero.

Note

The BSS component *must* be specified, contrary to the advice in the GDB User Guide.

The client then fetches the value of all the registers, so it can populate its register cache. It first specifies that operations such as these apply to all threads (**Hg-1** packet), then requests the value of all registers (**g** packet).

Finally the client offers to supply any symbolic data required by the server. In this example, no data is needed, so a reply of **"OK"** is sent.

Through this exchange, the GDB client shows the following output:

```
(gdb) target remote :51000
Remote debugging using :51000
0x00000100 in _start ()
(gdb)
```

3.2.2. The load Command

The RSP packet exchanges to implement the GDB **load** command are shown as a sequence diagram in Figure 3.2. In this example a program with a text section of 4752 (0x1290) bytes at address 0x0 and data section of 15 (0xe) bytes at address 0x1290 is loaded.

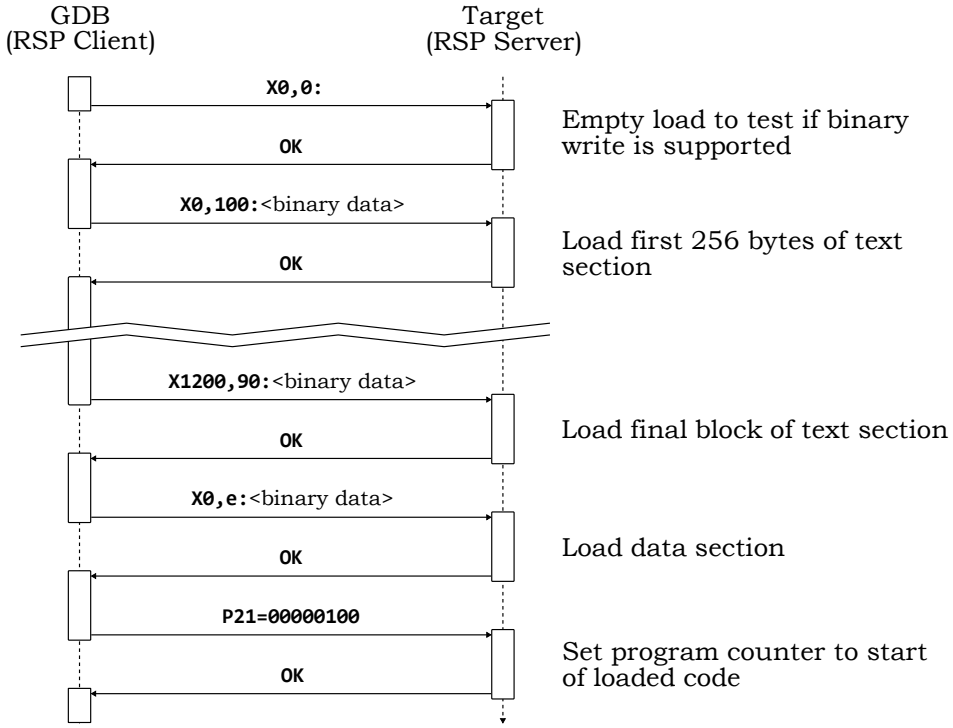

Figure 3.2. RSP packet exchanges for the GDB load command

The first packet is a binary write of zero bytes (**X0,0:**). A reply of **"OK"** indicates the target supports binary writing, an empty reply indicates that binary write is not supported, in which case the data will be loaded using **M** packets.

Note

This initial dialog is 7-bit clean, even though it uses the **X** packet. It can therefore safely be used with connections that are not 8-bit clean.

Caution

The use of a null reply to indicate that **X** packet transfers are not supported is not documented in the GDB User Guide.

Having established in this case that binary transfers are permitted, each section of the loaded binary is transmitted in blocks of up to 256 binary data bytes.

Had binary transfers not been permitted, the sections would have been transferred using **M** packets, using pairs of hexadecimal digits for each byte.

Finally the client sets the value of the program counter to the entry point of the code using a **P** packet. In this example the program counter is general register 33 and the entry point is address 0x100.

Through this exchange, the GDB client shows the following output:

```
(gdb) load hello
Loading section .text, size 0x1290 lma 0x0
Loading section .rodata, size 0xe lma 0x1290
Start address 0x100, load size 4766
Transfer rate: 5 KB/sec, 238 bytes/write.
(gdb)
```

3.2.3. Examining Registers

Examining registers in GDB causes no RSP packets to be exchanged. This is because the GDB client always obtains values for all the registers whenever it halts and caches that data. So for example in the following command sequence, there is no RSP traffic.

```
(gdb) print $pc
$1 = (void (*)()) 0x1264 <main+16>
(gdb)
```

3.2.4. Examining Memory

All GDB commands which involve examining memory are mapped by the client to a series of **m** packets. Unlike registers, memory values are *not* cached by the client, so repeated examination of a memory location will lead to multiple **m** packets for the same location.

The packet exchanges to implement the GDB **disassemble** command for a simple function are shown as a sequence diagram in Figure 3.3. In this example the **simputc ()** function is disassembled.

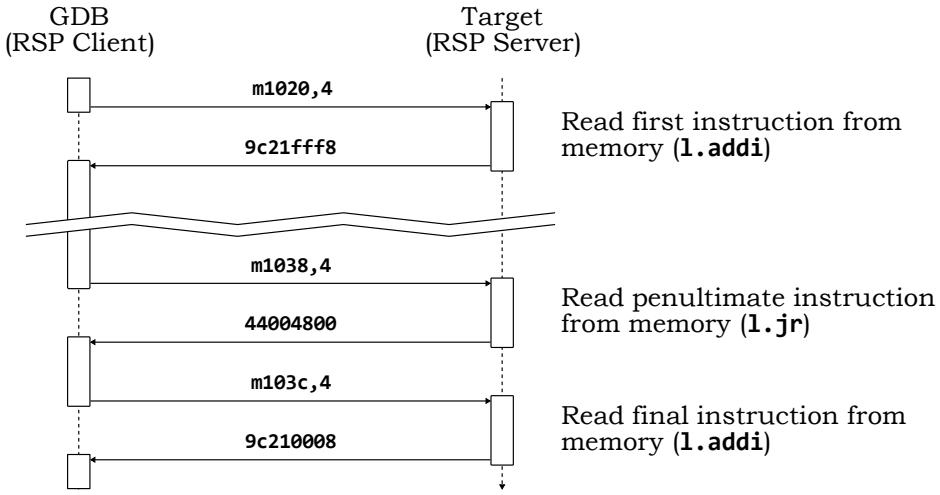

Figure 3.3. RSP packet exchanges for the GDB disassemble command

The **disassemble** command in the GDB client generates a series of RSP **m** packets, to obtain the instructions required one at a time.

Through this exchange, the GDB client shows the following output:

```
(gdb) disas simputc
Dump of assembler code for function simputc:
0x00001020 <simputc+0>: l.addi   r1,r1,-8
0x00001024 <simputc+4>: l.sw     0(r1),r2
0x00001028 <simputc+8>: l.addi   r2,r1,8
0x0000102c <simputc+12>:        l.sw     -4(r2),r3
0x00001030 <simputc+16>:        l.nop    4
0x00001034 <simputc+20>:        l.lwz    r2,0(r1)
0x00001038 <simputc+24>:        l.jr     r9
0x0000103c <simputc+28>:        l.addi   r1,r1,8
End of assembler dump.
(gdb)
```

3.2.5. The stepi Command

The RSP offers two mechanisms for stepping and continuing programs. The original mechanism has the thread concerned specified with a **Hc** packet, and then the thread stepped or continued with a **s**, **S**, **c** or **C** packet.

The newer mechanism uses the **vCont:** packet to specify the command and the thread ID in a single packet. The availability of the **vCont:** packet is established using the **vCont?** packet.

The simplest GDB execution command is the **stepi** command to step the target a single machine instruction. The RSP packet exchanges to implement the GDB **stepi** command are shown as a sequence diagram in Figure 3.4. In this example the instruction at address 0x100 is executed.

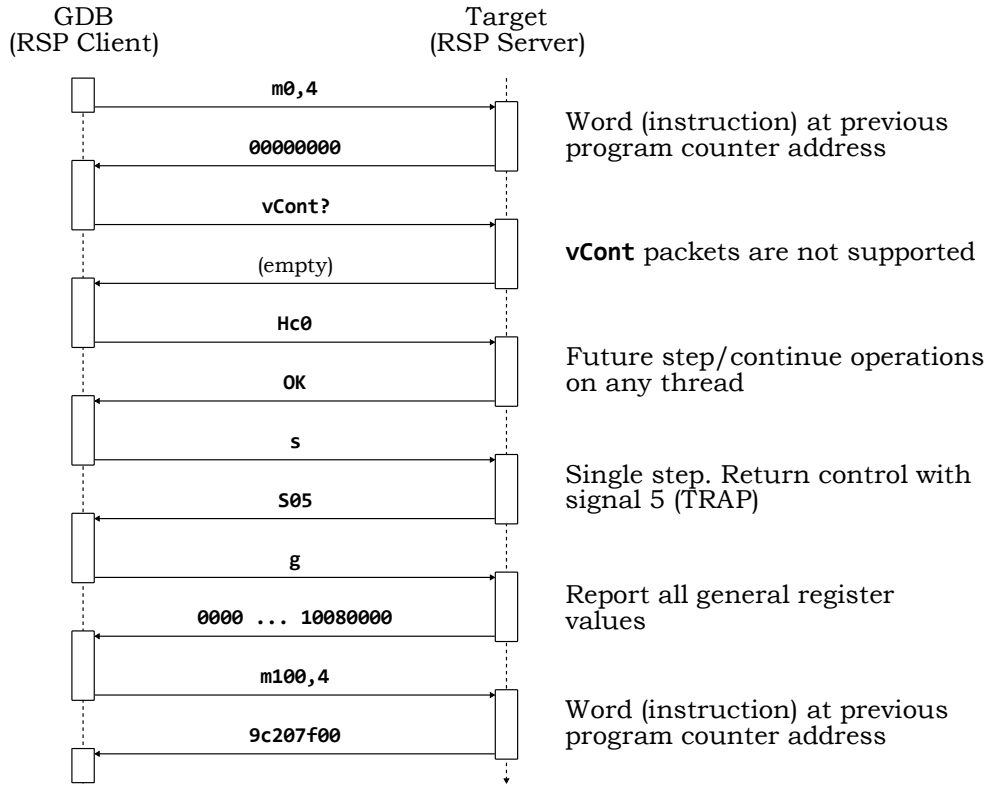

Figure 3.4. RSP packet exchanges for the GDB stepi command

The first exchange is related to the definition of the architecture used in this example. Before stepping any instruction, GDB needs to know if there is any special behavior due to this instruction occupying a delay slot. This is achieved by calling the **`gdbarch_single_step_through_delay ()`** function. In this example, that function reads the instruction at the previous program counter (in this case address 0x0) to see if it was an instruction with a delay slot. This is achieved by using the **m** packet to obtain the 4 bytes of instruction at that address.

The next packet, **vCont?** from the client seeks to establish if the server supports the **vCont** packet. A null response indicates that it is not.

Note

The **vCont?** packet is used only once, and the result cached by the GDB client. Subsequent step or continue commands will not result in this packet being reissued.

The client then establishes the thread to be used for the step with the **Hc0** packet. The value 0 indicates that any thread may be used by the server.

Note

Note the difference to the earlier use of the **Hc** packet (see Section 3.2.1), where a value of -1 was used to mean *all* threads.

Note

The GDB client remembers the thread currently in use. It does not issue further **Hc** packets unless the thread has to change.

The actual step is invoked by the **s** packet. This does not return a result to the GDB client until it has completed. The reply indicates that the server stopped for signal 5 (TRAP exception).

Caution

In the RSP, the **s** packet indicates stepping of a single machine instruction, *not* a high level statement. In this way it maps to GDB's **stepi** command, *not* its **step** command (which confusingly can be abbreviated to just **s**).

The last two exchanges are a **g** and **m** packet. These allow GDB to reload its register cache and note the instruction just executed.

Through this exchange, the GDB client shows the following output:

```
(gdb) stepi
0x00000104 in _start ()
(gdb)
```

3.2.6. The step Command

The GDB **step** command to step the target a single high level instruction is similar to the **stepi** instruction, and works by using multiple **s** packets. However additional packet exchanges are also required to provide information to be displayed about the high level data structures, such as the stack.

The RSP packet exchanges to implement the GDB **step** command are shown as a sequence diagram in Figure 3.5. In this example the first instruction of a C `main ()` function is executed.

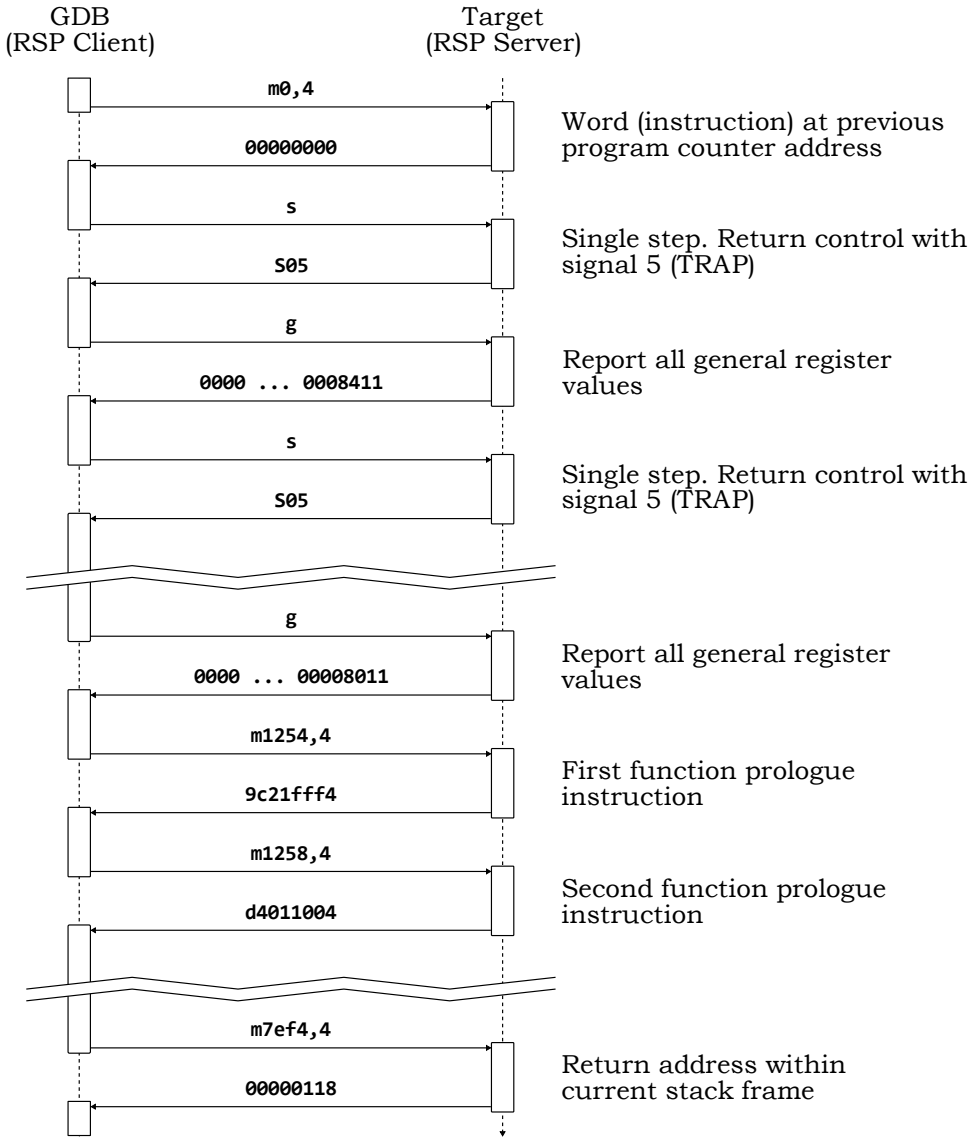

Figure 3.5. RSP packet exchanges for the GDB step command

The exchanges start similarly to the **stepi**, although, since this is not the first step, there are no **vCont?** or **Hc** packets.

The high level language step is mapped by the client GDB session into a series of **s** packets, after each of which the register cache is refreshed by a **g** packet.

After the step, are a series of reads of data words, using **m** packets. The first group are from the code. This is the first execution in a new function, and the frame analysis functions of the GDB client are analyzing the function prologue, to establish the location of key values (stack pointer, frame pointer, return address).

The second group access the stack frame to obtain information required by GDB. In this example the return address from the current stack frame.

Through this exchange, the GDB client shows the following output:

```
(gdb) step
main () at hello.c:41
```

```
41        simputs( "Hello World!\n" );
(gdb)
```

3.2.7. The cont Command

The packet exchange for the GDB **continue** is very similar to that for the **step** (see Section 3.2.6). The difference is that in the absence of a breakpoint, the target program may complete execution. A simple implementation need not trap the exit—GDB will handle the loss of connection quite cleanly.

The RSP packet exchanges to implement the GDB **continue** command are shown as a sequence diagram in Figure 3.6. In this example the target executes to completion and exits, without returning a reply packet to the GDB client.

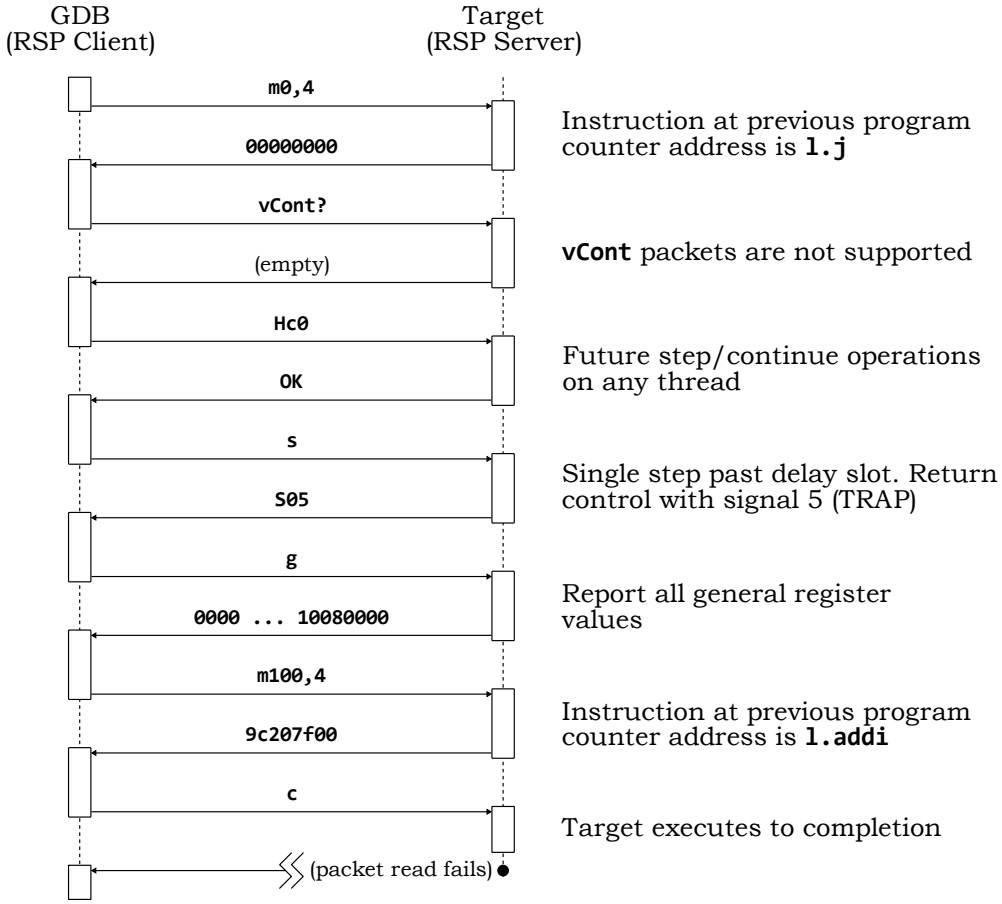

Figure 3.6. RSP packet exchanges for the GDB continue command

The packet exchange is initially the same as that for a GDB **step** or **stepi** command (see Figure 3.4).

In this example the `gdbarch_single_step_through_delay ()` function finds that the previously executed instruction is a jump instruction (**m** packet). Since the target may be in a delay slot, it executes a single step (**s** packet) to step past that slot, followed by notification of the TRAP exception (**S05** packet) and register cache reload (**g** packet).

The next call to `gdbarch_single_step_through_delay ()` determines that the previous instruction did not have a delay slot (**m** packet), so the **c** packet can be used to resume execution of the target.

Since the target exits, there is no reply to the GDB client. However it correctly interprets the loss of connection to the server as target execution. Through this exchange, the GDB client shows the following output:

```
(gdb) continue
Continuing.
Remote connection closed
(gdb)
```

3.2.8. The break Command

The GDB command to set breakpoints, **break** does not immediately cause a RSP interaction. GDB only actually sets breakpoints immediately before execution (for example by a **continue** or **step** command) and immediately clears them when a breakpoint is hit. This minimizes the risk of a program being left with breakpoints inserted, for example when a serial link fails.

The RSP packet exchanges to implement the GDB **break** command and a subsequent **continue** are shown as a sequence diagram in Figure 3.7. In this example a breakpoint is set at the start of the function **`simputs ()`**.

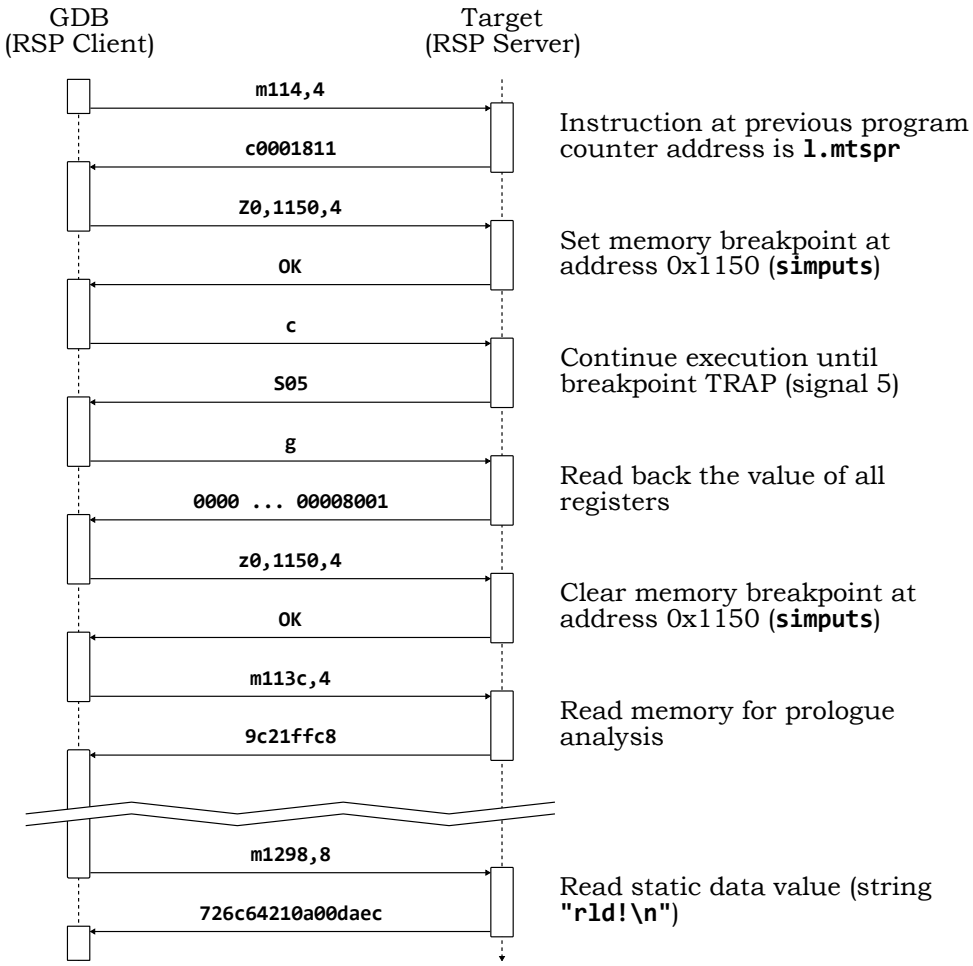

Figure 3.7. RSP packet exchanges for the GDB break and continue commands

The command sequence is very similar to that of the plain **continue** command (see Section 3.2.7). With two key differences.

First, immediately before the **c** packet, the breakpoint is set with a **Z0** packet. Secondly, as soon as the register cache has been refreshed (**g** packet) when control returns, the program counter is stepped back to re-execute the instruction at the location of the TRAP with a **P** packet and the breakpoint is cleared with a **z0** packet. In this case only a single breakpoint (at location 0x1150, the start of function **simputs ()**) is set. If there were multiple breakpoints, they would all be set immediately before the **c** packet and cleared immediately after the **g** packet.

In this example, the client ensures that the program counter is set to point to the TRAP instruction just executed, *not* the instruction following.

An alternative to adjusting the program counter in the target is to use the GDB architecture value **decr_pc_after_break ()** value to specify that the program counter should be wound back. In this case an additional **P** packet would be used to reset the program counter register. Whichever approach is used, it means that when execution resumes, the instruction which was replaced by a trap instruction will be executed first.

Note

Perhaps rather surprisingly, it is the responsibility of the target RSP server, not the GDB client to keep track of the substituted instructions.

Through this exchange, the GDB client shows the following output:

```
(gdb) break simputs
Breakpoint 1 at 0x1150: file utils.c, line 90.
(gdb) c
Continuing.

Breakpoint 1, simputs (str=0x1290 "Hello World!\n") at utils.c:90
90        for( i = 0; str[i] != '\0' ; i++ ) {
(gdb)
```

The example here showed the use of a memory breakpoint (also known as a software breakpoint). GDB also supports use of hardware watchpoints explicitly through the **hbreak** command. These behave analogously to memory breakpoints in RSP, but using **z1** and **Z1** packets.

If a RSP server implementation does not support hardware breakpoints it should return an empty packet to any request for insertion or deletion.

3.2.9. The watch Command

If hardware watchpoints are supported (the default assumption in GDB), then the setting and clearing of watchpoints is very similar to breakpoints, but using **z2** and **Z2** packets (for write watchpoints), **z3** and **Z3** packets (for read watchpoints) and **z4** and **Z4** packets (for access watchpoints)

GDB also supports software write watchpoints. These are implemented by single stepping the target, and examining the watched value after each step. This is painfully slow when GDB is running native. Under RSP, where each step involves an number of packet exchanges, the performance drops ever further. Software watchpointing should be restricted to the shortest section of code possible.

3.2.10. The detach and disconnect Commands

The rules for **detach** mandate that it breaks the connection with the target, and allows the target to resume execution. By contrast, the **disconnect** command simply breaks the connec-

tion. A reconnection (using the **target remote** command) should be able to resume debugging at the point where the previous connection was broken.

The **disconnect** command just closes the serial connection. It is up to the target server to notice the connection has broken, and to try to re-establish a connection.

The **detach** command requires a RSP exchange with the target for a clean shutdown. The RSP packet exchanges to implement the command are shown as a sequence diagram in Figure 3.8.

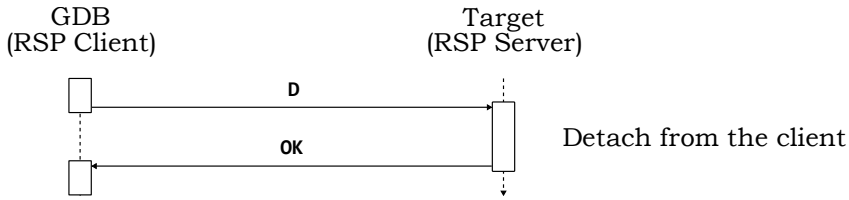

Figure 3.8. RSP packet exchanges for the GDB detach command

The exchange is a simple **D** packet to which the target responds with an **OK** packet, before closing the connection.

Through this exchange, the GDB client shows the following output:

```
(gdb) detach
Ending remote debugging.
(gdb)
```

The **disconnect** command has no dialog of itself. The GDB client shows the following output in a typical session. However there are no additional packet exchanges due to the disconnect.

```
(gdb) target remote :51000
Remote debugging using :51000
0x00000100 in _start ()
(gdb) load hello
Loading section .text, size 0x1290 lma 0x0
Loading section .rodata, size 0xe lma 0x1290
Start address 0x100, load size 4766
Transfer rate: 5 KB/sec, 238 bytes/write.
(gdb) break main
Breakpoint 1 at 0x1264: file hello.c, line 41.
(gdb) c
Continuing.

Breakpoint 1, main () at hello.c:41
41        simputs( "Hello World!\n" );
(gdb) disconnect
Ending remote debugging.
(gdb) target remote :51000
Remote debugging using :51000
main () at hello.c:41
41        simputs( "Hello World!\n" );
(gdb) c
Continuing.
Remote connection closed
```



```
(gdb)
```

Unlike with the **detach** command, when debugging is reconnected through **target remote**, the target is still at the point where execution terminated previously.

3.3. GDB Extended Remote Command Dialogs

The following sections show diagrammatically how various GDB commands map onto RSP packet exchanges to implement the desired behavior with extended remote debugging (i.e when connecting with **target extended-remote**).

3.3.1. The target extended-remote Command

The RSP packet exchanges to implement the GDB **target extended-remote** command are shown as a sequence diagram in Figure 3.9.

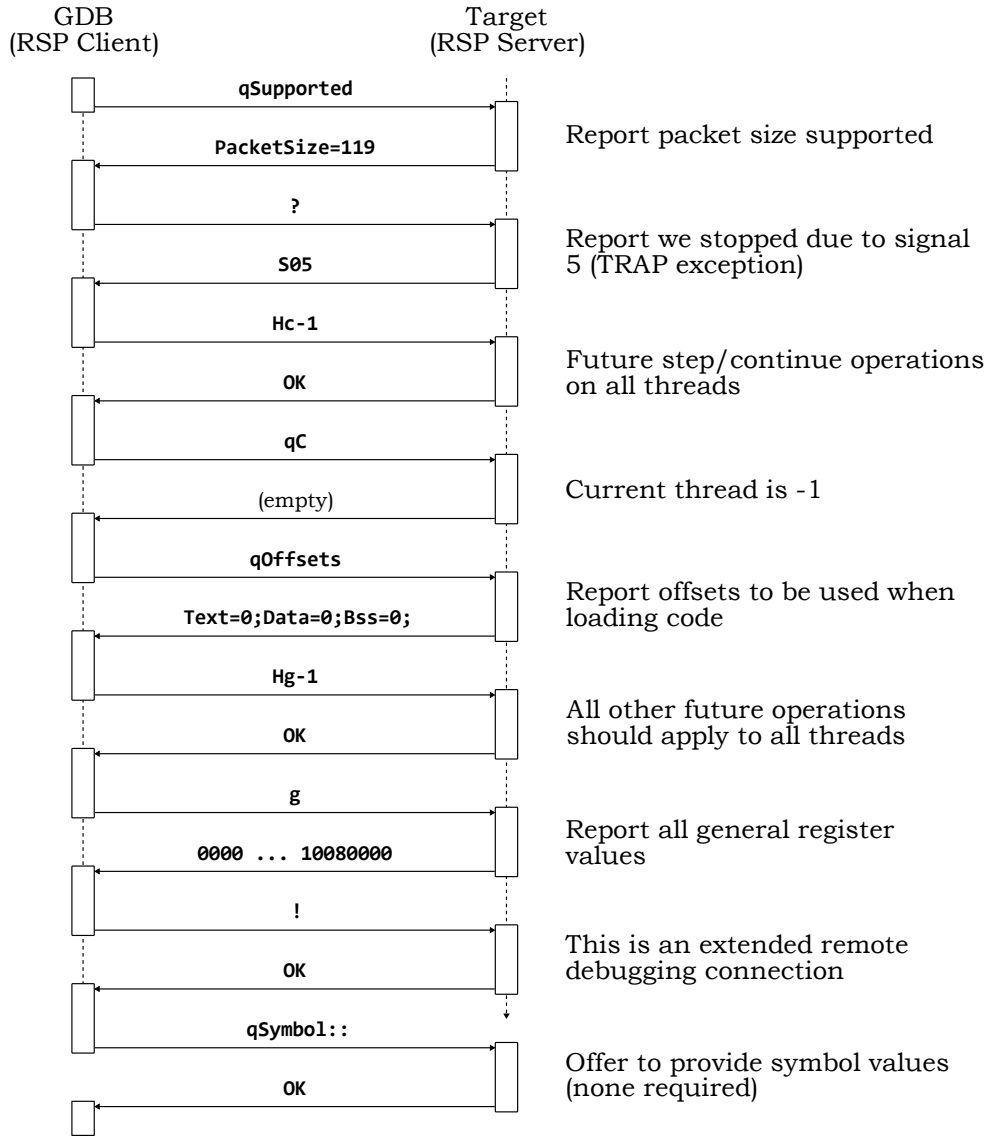

Figure 3.9. RSP packet exchanges for the GDB target remote command

The dialog is almost identical to that for standard remote debugging (see Section 3.2.1). The difference is the penultimate **!** packet, notifying the target that this is an extended remote connection.

Through this exchange, the GDB client shows the following output:

```
(gdb) target extended-remote :51000
Remote debugging using :51000
0x00000100 in _start ()
(gdb)
```

3.4. GDB Asynchronous Remote Command Dialogs

The dialogs for asynchronous debugging in general parallel their synchronous equivalents. The only differences are in those commands which can specify a particular thread to execute or stop.

Chapter 4. RSP Server Implementation Example

The examples used are based on the RSP server implementation for the OpenRISC 1000 architectural simulator, *Or1ksim*.

The target is "bare metal". There is no operating system infrastructure necessarily present. In this context, commands relating to threads or the file system are of no meaning and not implemented.

4.1. The OpenRISC 1000 Architectural Simulator, Or1ksim

4.1.1. The OpenRISC 1000 Architecture

The OpenRISC 1000 architecture defines a family of free, open source RISC processor cores. It is a 32 or 64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, scalability and versatility.

The OpenRISC 1000 is fully documented in its Architecture Manual [6].

From a debugging perspective, there are three data areas that are manipulated by the instruction set.

1. Main memory. A uniform address space with 32 or 64-bit addressing. Provision for separate or unified instruction and data and instruction caches. Provision for separate or unified, 1 or 2-level data and instruction MMUs.
2. General Purpose Registers (GPRs). Up to 32 registers, 32 or 64-bit in length.
3. Special Purpose Registers (SPRs). Up to 32 groups each with up to 2048 registers, up to 32 or 64-bit in length. These registers provide all the administrative functionality of the processor: program counter, processor status, saved exception registers, debug interface, MMU and cache interfaces, etc.

The Special Purpose Registers (SPRs) represent a challenge for GDB, since they represent neither addressable memory, nor have the characteristics of a register set (generally modest in number).

A number of SPRs are of particular significance to the GDB implementation.

- *Configuration registers*. The Unit Present register (SPR 1, **`UPR`**), CPU Configuration register (SPR 2, **`CPUCFGR`**) and Debug Configuration register (SPR 7, **`DCFGR`**) identify the features available in the particular OpenRISC 1000 implementation. This includes the instruction set in use, number of general purpose registers and configuration of the hardware debug interface.
- *Program counters*. The Previous Program Counter (SPR 0x12, **`PPC`**) is the address of the instruction just executed. The Next Program Counter (SPR 0x10, **`NPC`**) is the address of the next instruction to be executed. The **`NPC`** is the value reported by GDBs **$pc** variable.
- *Supervision Register*. The supervision register (SPR 0x11, **`SR`**) represents the current status of the processor. It is the value reported by GDBs status register variable, **$ps**.

4.1.2. The OpenRISC 1000 Debug Unit

Of particular importance are the SPRs in group 6 controlling the debug unit (if present). The debug unit can trigger a *trap* exception in response to any one of up to 10 *watchpoints*. Watch-

points are logical expressions built by combining *matchpoints*, which are simple point tests of particular behavior (has a specified address been accessed for example).

- *Debug Value and Control registers*. There are up to 8 pairs of Debug Value (SPR 0x3000–0x3007, **DVR0** through **DVR7**) and Debug Control (SPR 0x3008–0x300f, **DCR0** through **DCR7**) registers. Each pair is associated with one hardware *matchpoint*. The Debug Value register in each pair gives a value to compare against. The Debug Control register indicates whether the matchpoint is enabled, the type of value to compare against (instruction fetch address, data load and/or store address data load and/or store value) and the comparison to make (equal, not equal, less than, less than or equal, greater than, greater than or equal), both signed and unsigned. If the matchpoint is enabled and the test met, the corresponding matchpoint is triggered.
- *Debug Watchpoint counters*. There are two 16-bit Debug Watchpoint Counter registers (SPR 0x3012–0x3013, **DWCR0** and **DWCR1**), associated with two further matchpoints. The upper 16 bits are a value to match, the lower 16 bits a counter. The counter is incremented when specified matchpoints are triggered (see Debug Mode register 1). When the count reaches the match value, the corresponding matchpoint is triggered.

  

  Caution

  There is potential ambiguity in that counters are incremented in response to matchpoints and also generate their own matchpoints. It is not good practice to set a counter to increment on its own matchpoint!
- *Debug Mode registers*. There are two Debug Mode registers to control the behavior of the the debug unit (SPR 0x3010–0x3011, **DMR1** and **DMR2**). **DMR1** provides a pair of bits for each of the 10 matchpoints (8 associated with DVR/DCR pairs, 2 associated with counters). These specify whether the watchpoint is triggered by the associated matchpoint, by the matchpoint AND-ed with the previous watchpoint or by the matchpoint OR-ed with the previous watchpoint. By building chains of watchpoints, complex logical tests of hardware behavior can be built up.
  Two further bits in **DMR1** enable single step behavior (a trap exception occurs on completion of each instruction) and branch step behavior (a trap exception occurs on completion of each branch instruction).

  DMR2 contains an enable bit for each counter, 10 bits indicating which watchpoints are assigned to which counter and 10 bits indicating which watchpoints generate a trap exception. It also contains 10 bits of output, indicating which watchpoints have generated a trap exception.
- *Debug Stop and Reason registers*. In normal operation, all OpenRISC 1000 exceptions are handled through the exception vectors at locations 0x100 through 0xf00. The Debug Stop register (SPR 0x3014, **DSR**) is used to assign particular exceptions instead to the JTAG interface. These exceptions stall the processor, allowing the machine state to be analyzed through the JTAG interface. Typically a debugger will enable this for trap exceptions used for breakpointing.
  Where an exception has been diverted to the development interface, the Debug Reason register (SPR 0x3021, **DRR**) indicates which exception caused the diversion. Note that although single stepping and branch stepping cause a trap, if they are assigned to the JTAG interface, they *do not* set the **TE** bit in the **DRR**. This allows an external debugger to distinguish between breakpoint traps and single/branch step traps.

4.1.3. The OpenRISC 1000 JTAG Interface

In a physical OpenRISC 1000 chip, debugging would be via the JTAG interface. However since the examples used here are based on the architectural simulator, the JTAG interface is not described further here.

4.1.4. Application Binary Interface (ABI)

The ABI for the OpenRISC 1000 is described in Chapter 16 of the Architecture Manual [6]. However the actual GCC compiler implementation differs very slightly from the documented ABI. Since precise understanding of the ABI is critical to GDB, those differences are documented here.

- Register Usage: R12 is used as another callee-saved register. It is never used to return the upper 32 bits of a 64-bit result on a 32-bit architecture. All values greater than 32-bits are returned by a pointer.
- Although the specification requires stack frames to be *double* word aligned, the current GCC compiler implements *single* word alignment.
- Integral values more than 32 bits (64 bits on 64-bit architectures), structures and unions are returned as pointers to the location of the result. That location is provided by the *calling* function, which passes it as a first argument in GPR 3. In other words, where a function returns a result of this type, the first true argument to the function will appear in R4 (or R5/R6 if it is a 64-bit argument on a 32-bit architecture).

4.1.5. Or1ksim: the OpenRISC 1000 Architectural Simulator

Or1ksim is an instruction set simulator (ISS) for the OpenRISC 1000 architecture. At present only the 32-bit architecture is modeled. In addition to modeling the core processor, *Or1ksim* can model a number of peripherals, to provide the functionality of a complete System-on-Chip (SoC).

Or1ksim implements the RSP server side. It is the implementation of this RSP server which forms the example for this application note.

4.2. OpenRISC 1000 GDB Architectural Specification

The GDB architectural specification (**`gdbarch`**) for OpenRISC 1000 is fully documented in Embecosm Application Note 3 ([2]). This section notes some important features, which will be of relevance to the RSP server implementation.

- All data sizes are specified to match the ABI for the OpenRISC 1000
- All memory breakpoints are implemented at the program counter using the **`l.trap 1`** opcode, which like all OpenRISC 1000 instructions is 4 bytes long.
  This means that after a trap due to a breakpoint, the program counter must be stepped back, to allow re-execution on resumption of the instruction that was replaced by **`l.trap`**
- A total of 35 registers are defined to GDB: The 32 general purpose registers, the previous program counter, the next program counter (colloquially known as *the* program counter) and the supervision register. There are no pseudo-registers.

4.3. Overview of the RSP Server Implementation

All the code for the OpenRISC 1000 RSP server interface can be found in **`debug/rsp-server.c`**. The interface is specified in the header file, **`debug/rsp-server.h`**.

The code is commented for post-processing with **`doxygen`** ([5]).

4.3.1. External Code Interface

The external interface to the RSP server code is through three **`void`** functions.

1. **rsp_init ()**. Called at start up to initialize the RSP server. It initializes global data structures (discussed in Section 4.3.2) and then sets up a TCP/IP listener on the configured RSP port.
2. **handle_rsp ()**. Called repeatedly when the processor is stalled to read packets from any GDB client and process them.
3. **rsp_exception ()**. Called from the simulator to record any exceptions that occur, for subsequent use by **handle_rsp ()**. It takes a single argument, the OpenRISC 1000 exception handler entry address, which is mapped by the RSP server to the equivalent GDB target signal.

4.3.2. Global Data Structures

The RSP server has one data structure, **rsp**, shared amongst its implementing functions (and is thus declared **static** in **rsp-server.c**).

```
static struct
{
  int                client_waiting;
  int                proto_num;
  int                server_fd;
  int                client_fd;
  int                sigval;
  unsigned long int  start_addr;
  struct mp_entry   *mp_hash[MP_HASH_SIZE];
} rsp;
```

The fields are:

- **client_waiting**. A flag to indicate if the target has previously been set running (by a GDB **continue** or **step**) instruction, in which case the client will be waiting for a response indicating when and why the server has stopped.
- **proto_num**. The number of the communication protocol used (in this case TCP/IP).
- **server_fd**. File handle of the server connection to the RSP port, listening for connections. Set to -1 if it is not open.
- **client_fd**. File handle of the current client connection to the RSP port, on which all packet transfers take place. Set to -1 if it is not open.
- **sigval**. The last exception raised by the target as a GDB target signal number. Set by the simulator calling **rsp_exception ()**.
- **start_addr**. The start address of the last run. Needed to support the restart function of extended remote debugging.
- **mp_hash**. Pointer to the hash table of matchpoints set (see Section 4.3.2.1).

The RSP server also draws on several *Or1ksim* data structures. Most notably **config** for configuration data and **cpu_state** for all the CPU state data.

4.3.2.1. The Matchpoint Hash Table

The matchpoint hash table is implemented as an open hash table, where the hash table entry is calculated as the address of the matchpoint modulo the size of the hash table (**MP_HASH_SIZE**)

and the key is formed from the address and the matchpoint type. Matchpoint types are defined for memory and hardware breakpoints and hardware write, read and access watchpoints:

```
enum mp_type {
  BP_MEMORY   = 0,
  BP_HARDWARE = 1,
  WP_WRITE    = 2,
  WP_READ     = 3,
  WP_ACCESS   = 4
};
```

Each entry in the table holds the instruction at the location of the matchpoint, which in the case of memory breakpoints will have been replaced by **l.trap**

```
struct mp_entry
{
  enum mp_type       type;
  unsigned long int  addr;
  unsigned long int  instr;
  struct mp_entry   *next;
};
```

Linking through the **next** field allows multiple entries with the same hash value to be stored.

Interface to the hash table is through four functions:

- **mp_hash_init ()**. **void** function which sets all the hash table slots to **NULL**
- **mp_hash_add ()**. **void** function which adds an entry to the hash table (if it is not already there). It takes three arguments, the matchpoint type and address and the instruction stored at that address. Repeated adding of the same entry has no effect, which provides convenient behavior for debugging over noisy connections where packets may be duplicated.
- **mp_hash_lookup ()**. Function to look up a key in the hash table. It takes a matchpoint type and address and returns a pointer to the entry (as a pointer to **struct mp_entry**) or **NULL** if the key is not found.
- **mp_hash_delete ()**. Function with the same behavior as **mp_hash_lookup ()**, but also deletes the entry from the hash table if it is found there. If the entry is not found, it silently does nothing (and returns **NULL**).

Note

This function returns a pointer to the **struct–mp_entry** deleted from the hash table if the key is found. To avoid memory leaks it is important that the caller delete this structure (using **free ()**) when the data has been extracted.

4.3.3. Top Level Behavior

The RSP server initialization, **rsp_init ()** is called from the main simulator initialization, **sim_init ()** in **toplevel-support.c**.

The main simulation initialization is also modified to start the processor stalled on a TRAP exception if RSP debugging is enabled. This ensures that the handler will be called initially.

The main loop of *Or1ksim*, called after initialization, is in the function **exec_main ()** in **cpu/or32/execute.c**.

If RSP debugging is enabled in the *Or1ksim* configuration, the code to interact with the RSP client (**handle_rsp ()**) is called at the start of each iteration, *but only if the processor is stalled*. The handler is called repeatedly until an interaction with the client unstalls the processor (i.e. a **step** or **continue** function.

```
void
exec_main ()
{
  long long time_start;

  while (1)
    {
      time_start = runtime.sim.cycles;
      if (config.debug.enabled)
        {
          while (runtime.cpu.stalled)
            {
              if (config.debug.rsp_enabled)
                {
                  handle_rsp ();
                }

              ...
```

Since interaction with the client can only occur when the processor is stalled, BREAK signals (i.e. ctrl-C) cannot be intercepted.

It would be possible to poll the connection on *every* instruction iteration, but the performance overhead on the simulator would be unacceptable.

An implementation to pick up BREAK signals should use event driven I/O - i.e. with a signal handler for **SIGIO**. An alternative is to poll the interface less frequently when the CPU is not stalled. Since *Or1ksim* executes at several MIPS, polling every 100,000 cycles would mean a response to ctrl-C of less than 100ms, while adding no significant overhead.

4.3.3.1. Exception handling

The RSP interface will only pick up those exceptions which cause the processor to stall. These are the exceptions routed to the debug interface, rather than through their exception vectors, and are specified in the Debug Stop Register (set during initialization). In the present implementation, only TRAP exceptions are picked up this way, allowing the debugger to process memory based breakpoints. However an alternative implementation could allow the debugger to see all exceptions.

Exceptions will be processed at the start of each iteration by **handle_rsp ()**. However the handler needs to know which signal caused the exception. This is achieved by modifying the main debug unit exception handling function (**debug_ignore_exception ()** in **debug/debug-unit.c**) to call **rsp_exception ()** if RSP is enabled for any exception handled by the debug unit. This function stores the exception (translated to a GDB target signal) in **rsp.sigval**.


```
int
debug_ignore_exception (unsigned long  except)
{
  int           result = 0;
  unsigned long dsr    = cpu_state.sprs[SPR_DSR];

  switch (except)
    {
    case EXCEPT_RESET:    result = (dsr & SPR_DSR_RSTE);  break;
    case EXCEPT_BUSERR:   result = (dsr & SPR_DSR_BUSEE); break;

  ...

  cpu_state.sprs[SPR_DRR] |= result;
  set_stall_state (result != 0);

  if (config.debug.rsp_enabled && (0 != result))
    {
      rsp_exception (except);
    }

  return  (result != 0);

}       /* debug_ignore_exception () */
```

For almost all exceptions, this approach is suitable. However TRAP exceptions due to single stepping are taken at the end of each instruction execution and do not use the standard exception handling mechanism.

The **exec_main ()** function already includes code to handle this towards the end of the main loop. This is extended with a call to **rsp_exception ()** if RSP debugging is enabled.

```
      if (config.debug.enabled)
        {
          if (cpu_state.sprs[SPR_DMR1] & SPR_DMR1_ST)
            {
              set_stall_state (1);

              if (config.debug.rsp_enabled)
                {
                  rsp_exception (EXCEPT_TRAP);
                }
            }
        }
```

4.4. The Serial Connection

4.4.1. Establishing the Server Listener Socket

A TCP/IP socket to listen on the RSP port is created in **rsp_init ()**, and its file descriptor stored in **rsp.server_fd**. As a variant, if the port is configured to be 0, the socket uses the port specified for the **or1ksim-rsp** service.

The setup uses standard POSIX calls to establish the socket and associate it with a TCP/IP port. The interface is set to be non-blocking and marked as a passive port (using a call to **`listen ()`**), with at most one outstanding client request. There is no meaning to the server handling more than one client GDB connection.

The main RSP handler function **`handle_rsp ()`** checks that the server port is still open. This may be closed if there is a serious error. In the present implementation, **`handle_rsp ()`** gives up at this point, but a richer implementation could try reopening a new server port.

4.4.2. Establishing the Client Connection

If a client connection is yet to be established, then **`handle_rsp ()`** blocks until a connection request is made. A valid request is handled by **`rsp_server_request ()`**, which opens a connection to the client, saving the file descriptor in **`rsp.client_fd`**.

This connection is also non-blocking. Nagel's algorithm is also disabled, since all packet bytes should be sent immediately, rather than being queued to build larger blocks.

4.4.3. Communicating with the Client

Having established a client connection if necessary, **`handle_rsp ()`** blocks until packet data is available. It then calls **`rsp_client_request ()`** to read the packet, provide the required behavior and generate any reply packets.

4.5. The Packet Interface

4.5.1. Packet Representation

Although packets are character based, they cannot simply be represented as strings, since binary packets may contain the end of string character (zero). Packets are therefore represented as a simple **`struct`**, **`rsp_buf`**:

```
struct rsp_buf
{
  char  data[GDB_BUF_MAX];
  int   len;
};
```

For convenience, all packets have a zero added at location **`data[len]`**, allowing the data field of non-binary packets to be printed as a simple string for debugging purposes.

4.5.2. Getting Packets

The packet reading function is **`get_packet ()`**. It looks for a well formed packet, beginning with '$', with '#' at the end of data and a valid 2 byte checksum (see Figure 2.2 in Section 2.3 for packet representation details).

If a valid packet is found, '+' is returned using **`put_rsp_char ()`** (see Section 4.5.3.1) and the packet is returned as a pointer to a **`struct rsp_buf`**. Otherwise '-' is returned and the loop repeated to get a new packet (presumably retransmitted by the client).

The buffer is statically allocated within **`get_packet ()`**. This is acceptable, since two received packets cannot be in use simultaneously.

In general errors are silently ignored (the connection could be poor quality). However bad checksums are noted in a warning message. In the event of end of file being encountered, **`get_packet ()`** returns immediately with **`NULL`** as result.

4.5.2.1. Character Input

The individual characters are read using **`get_rsp_char ()`**. The result is returned as an **`int`**, allowing -1 to be used to indicate end of file, or other error. In the event of end of file, or error, the client connection is closed and **`rsp.client_fd`** set to -1.

4.5.3. Sending Packets

The packet writing function is **`put_packet ()`**. It takes as argument a **`struct rsp_buf`** and creates a well formed packet, beginning with '$', with '#' at the end of data and a valid 2 byte checksum (see Figure 2.2 in Section 2.3 for packet representation details).

The acknowledgment character is read using **`get_rsp_char ()`** (see Section 4.5.2.1). If successful ('+'), the function returns. Otherwise the packet is repeatedly resent until ('+') is received as a response.

Errors on writing are silently ignored. If the read of the acknowledgment returns -1 (indicating failure of the connection or end-of-file), **`put_packet ()`** returns immediately.

4.5.3.1. Character Output

The individual characters are written by **`put_packet ()`** using **`put_rsp_char ()`**. In the event of an error other than a retry request or interrupt a warning is printed and the connection closed.

4.6. Convenience Functions

A number of convenience functions are provided for RSP protocol behavior that is repeatedly required.

4.6.1. Convenience String Packet Output

Many response packets take the form of a fixed string. As a convenience **`put_str_packet ()`** is provided. This takes a constant string argument, from which a **`struct rsp_buf`** is constructed. This is then sent using **`put_packet ()`**.

4.6.2. Conversion Between Binary and Hexadecimal Characters

The function **`hex ()`** takes an ASCII character which is a valid hexadecimal digit (upper or lower case) and returns its value (0-15 decimal). Any invalid digit returns -1.

The static array **`hexchars[]`** declared at the top level in **`rsp-server.c`** provides a mapping from a hexadecimal digit value (in the range 0-15 decimal) to its ASCII character representation.

4.6.3. Conversion Between Binary and Hexadecimal Character Registers

For several packets, register values must be represented as strings of characters in target endian order. For convenience, the functions **`reg2hex ()`** and **`hex2reg ()`** are provided. Each takes a pointer to a buffer for the characters. For **`reg2hex ()`** a value to be converted is passed. For **`hex2reg ()`** the value represented is returned as a result.

4.6.4. Data "Unescaping"

The function **rsp_unescape ()** takes a pointer to a data buffer and a length and "unescapes" the buffer in place. The length is the size of the data *after* all escape characters have been removed.

4.6.5. Setting the Program Counter

The program counter (i.e. the address of the next instruction to be executed) is held in Special Purpose Register 16 (next program counter). Within *Or1ksim* this is cached in **cpu_state.pc**.

When changing the next program counter in *Or1ksim* it is necessary to change associated data which controls the delay slot pipeline. If there is a delayed transfer, the flag **cpu_state.delay_insn** is set. The address of the next instruction to be executed (which is affected by the delay slot) is held in the global variable, **pc_next**.

The utility function **set_npc ()** takes an address as argument. If that address is different to the current value of NPC, then the NPC (in **cpu_state.pc**) is updated to the new address, the delay slot pipeline is cleared (**cpu_state.delay_insn** is set to zero) and the following instruction (**pcnext**) is set to **cpu_state.pc**+4.

4.7. High Level Protocol Implementation

The high level protocol is driven from the function **rsp_client_request ()**, which is called from **handle_rsp ()** once a client connection is established.

This function calls **get_packet ()** to get the next packet from the client, and then switches on the first character of the packet data to determine the action to be taken.

The following sections discuss the implementation details of the various packet types that must be supported.

4.7.1. Deprecated Packets

Packets requesting functionality that is now deprecated are ignored (possibly with an error response if that is expected) and a warning message printed. The packets affected are: **b** (set baud rate), **B** (set a breakpoint), **d** (disable debug) and **r** (reset the system).

In each case the warning message indicates the recommended way to achieve the desired functionality.

4.7.2. Unsupported Packets

The development of an interface such as RSP can be incremental, where functionality is added in stages. A number of packets are not supported. In a few cases this is because the functionality is meaningless for the current target, but in the majority of cases, the functionality can be supported as the server is developed further in the future.

The unsupported packets are:

- **A**. Specifying the arguments for a program is hard on "bare metal". It requires determining whether the code has yet entered its **main ()** function and if not patching in pointers to the new arguments.
- **C** and **S**. Continuing or stepping with a signal is currently not supported. Implementing this would require insertion of an exception, which is not difficult, so this will be an enhancement for the near future.

- **F**. File I/O is not meaningful with a bare metal target, where a file-system may not be present.
- **i** and **I**. The target is an architectural simulator, executing one instruction at a time. So cycle accurate stepping is not available.
- **t**. The meaning (or use) of the search command is not clear, so this packet is not currently implemented.

4.7.3. Simple Packets

Some packets are very simple to handle, either requiring no response, or a simple fixed text response.

- **!**. A simple reply of **"OK"** indicates the target will support extended remote debugging.
- **D**. The detach is acknowledged with a reply packet of **"OK"** *before* the client connection is closed and **rsp.client_fd** set to -1. The semantics of detach require the target to resume execution, so the processor is unstalled using **set_stall_state (0)**.
- **H**. This sets the thread number of subsequent operations. Since thread numbers are of no relevance to this target, a response of **"OK"** is always acceptable.
- **k**. The kill request is used in extended mode before a restart or request to run a new program (**vRun** packet). Since the CPU is already stalled, it seems to have no additional semantic meaning. Since it requires no reply it can be silently ignored.
- **T**. Since this is a bare level target, there is no concept of separate threads. The one thread is always active, so a reply of **"OK"** is always acceptable.

4.7.4. Reporting the Last Exception

The response to the **?** packet is provided by **rsp_report_exception ()**. This is always a **S** packet. The signal value (as a GDB target signal) is held in **rsp.sigval**, and is presented as two hexadecimal digits.

4.7.5. Continuing

The **c** packet is processed by **rsp_continue ()**. Any address from which to continue is broken out from the packet using **sscanf ()**. If no address is given, execution continues from the current program counter (in **cpu_state.pc**).

The continue functionality is provided by the function **rsp_continue_generic ()** which takes an address and an *Or1ksim* exception as arguments, allowing it to be shared with the processing of the **C** packet (continue with signal) in the future. For the **c** packet, **EXCEPT_NONE** is used.

rsp_continue_generic () at present ignores its exception argument (the **C** packet is not supported). It sets the program counter to the address supplied using **set_npc ()** (see Section 4.6.5).

The control registers of the debug unit must then be set appropriately. The Debug Reason Register and watchpoint generation flags in Debug Mode Register 2 are cleared. The Debug Stop Register is set to trigger on TRAP exceptions (so memory breakpoints are picked up), and the single step flag is cleared in Debug Mode Register 1.

```
    cpu_state.sprs[SPR_DRR]   = 0;
    cpu_state.sprs[SPR_DMR2] &= ~SPR_DMR2_WGB;
    cpu_state.sprs[SPR_DMR1] &= ~SPR_DMR1_ST;
```

```
    cpu_state.sprs[SPR_DSR]  |= SPR_DSR_TE;
```

The processor is then unstalled (**set_stall_state (0)**) and the client waiting flag (**rsp.client_waiting**) set. This latter means that when **handle_rsp ()** is next entered, it will know that a reply is outstanding, and return the appropriate stop packet required when the processor stalls after a **continue** or **step** command.

4.7.6. Reading and Writing All Registers

The **g** and **G** packets respectively read and write all registers, and are handled by the functions **rsp_read_all_regs ()** and **rsp_write_all_regs ()**.

4.7.6.1. Reading All Registers

The register data is provided in a reply packet as a stream of hexadecimal digits for each register in GDB register order. For the OpenRISC 1000 this is the 32 GPRs followed by the Previous Program Counter, Next Program Counter and Supervision Register SPRs. Each register is presented in target endian order, using the convenience function **reg2hex ()**.

4.7.6.2. Writing All Registers

The register data follows the **G** as a stream of hexadecimal digits for each register in GDB register order. For the OpenRISC 1000 this is the 32 GPRs followed by the Previous Program Counter, Next Program Counter and Supervision Register SPRs. Each register is supplied in target endian order and decoded using the utility function **hex2reg ()**.

The corresponding values are set in the *Or1ksim* data structures. For the GPRs this is in the **cpu_state.regs** array. For the Previous Program Counter and Supervision Register it is the relevant entry in the **cpu_state.sprs** array. The Next Program Counter is set using the **set_npc ()** convenience function (see Section 4.6.5), which ensures associated variables, controlling the delay pipeline are also updated appropriately.

4.7.7. Reading and Writing Memory

The **m** and **M** packets respectively read and write blocks of memory, with the data represented as hexadecimal characters. The processing is provided by **rsp_read_mem ()** and **rsp_write_mem ()**.

4.7.7.1. Reading Memory

The syntax of the packet is **m,addr,len:**. **sscanf ()** is used to break out the address and length fields (both in hex).

The reply packet is a stream of bytes, starting from the lowest address, each represented as a pair of hex characters. Each byte is read from memory using the simulator function **eval_direct8 ()**, having first verified the memory area is valid using **verify_memoryarea ()**.

The packet is only sent if all bytes are read satisfactorily. Otherwise an error packet, **"E01"** is sent. The actual error number does not matter—it is not used by the client.

Caution

The use of **eval_direct8 ()** is not correct, since it ignores any caching or memory management. As a result the current implementation is only correct for configurations with no MMU or cache.

4.7.7.2. Writing Memory

The syntax of the packet is **m,addr,len:** followed by the data to be written as a stream of bytes, starting from the lowest address, each represented as a pair of hex characters. `sscanf ()` is used to break out the address and length fields (both in hex).

Each byte is written to memory using `set_program8 ()` (which ignores any read only constraints on the modeled memory), having first verified that the memory address is valid using `verify_memoryarea ()`.

If all bytes are written successfully, a reply packet `"OK"` is sent. Otherwise an error reply, `"E01"` is sent. The actual error number does not matter—it is not used by the client.

Caution

The use of `set_program8 ()` is not correct, since it ignores any caching or memory management. As a result the current implementation is only correct for configurations with no MMU or cache.

4.7.8. Reading and Writing Individual Registers

The **p** and **P** packets are implemented respectively by `rsp_read_reg ()` and `rsp_write_reg ()`.

These functions are very similar in implementation to `rsp_read_all_regs ()` and `rsp_write_all_regs ()` (see Section 4.7.6).

The two differences are that the packet data must be parsed to identify the register affected, and (clearly) only one register is read or written.

4.7.9. Query Packets

Query packets all start with **q**. The functionality is all provided in the function `rsp_query ()`.

4.7.9.1. Deprecated Query Packets

The **qL** and **qP** packets to obtain information about threads are now obsolete, and are ignored with a warning. An empty reply (meaning not supported) is sent to each.

These packets have been replaced by **qC**, **qfThreadInfo**, **qsThreadInfo**, **qThreadExtraInfo** and **qGetTLSAddr** packets (see Section 4.7.9.3).

4.7.9.2. Unsupported Query Packets

A number of query packets are not needed in an initial implementation, or make no sense for a "bare metal" target.

- **qCRC**. This can be implemented later by writing the code to compute a CRC for a memory area. A warning is printed and an error packet ("E01") returned.
- **qGetTLSAddr**. This is a highly operating system dependent function to return the location of thread local storage. It has no meaning in a simple "bare metal" target. An empty reply is used to indicate that the feature is not supported.
- **qRcmd**. This packet is used to run a remote command. Although this does not have a direct meaning, it is a useful way of passing arbitrary requests to the target. In the current implementation two commands **readspr** and **writespr** are provided to read and write values from and to Special Purpose Registers (needed for the GDB **info spr** and **spr** commands). These commands cannot be implemented using the main packets, since SPRs do not appear in either the memory map or the register file.

A side effect of this mechanism is that the remote commands are directly visible to the user through the GDB **monitor** command. Thus there are two ways to view a SPR. The "official" way:

```
(gdb) info spr npc
SYS.NPC = SPR0_16 = 256 (0x100)
(gdb)
```

And the unofficial way:

```
(gdb) monitor readspr 10
     100(gdb)
```

For this reason, defining and using a new **qXfer** packet type (see below) might be preferred as a way of accessing custom information such as SPR values.

- **qXfer:**. This packet is used to transfer "special" data to and from the target. A number of variants are already defined, to access particular features, some specific to certain targets and operating systems.
  This is the alternative way to provide SPR access, by providing a new variant **qXfer** specific to the OpenRISC 1000. However any new **qXfer** does demand integration within GDB.

  qXfer functionality must be specifically enabled using the **qSupported** packet (see Section 4.7.9.5). For the present this is not provided.

4.7.9.3. Queries About Threads

Although threads are not meaningful on the "bare metal" target, sensible replies can be given to most of the thread related queries by using -1 to mean "all threads".

- **qC**. An empty reply is used, which is interpreted as "use the previously selected thread". Since no thread is ever explicitly selected by the target, this will allow the client GDB session to use its default `NULL` thread, which is what is wanted.
- **qfThreadInfo** and **qsThreadInfo**. These packets are used to report the currently active threads. **qfThreadInfo** is used to report the first set of information and **qsThreadInfo** for all subsequent information, until a reply packet of `"l"` indicates the last packet.
  In this implementation, a reply packet of `"m-1"` (all packets are active is used for **qfThreadInfo** and a reply packet of `"l"` is used for **qsThreadInfo** to indicate there is no more information.
- **qThreadExtraInfo**. This should return a printed string, encoded as ASCII characters as hexadecimal digits with attributes of the thread specified as argument.
  The argument is always ignored (this target only has one thread), and the reply `"Runnable"` is sent back

4.7.9.4. Query About Executable Relocation

The **qOffsets** packet requests a reply string of the format `"Text=xx;Data=yy;Bss=zz"` to identify the offsets used in relocating the sections of code to be downloaded.

No relocation is used in this target, so the fixed string `"Text=0;Data=0;Bss=0"` is sent as a reply.

Caution

The GDB User Guide ([3]) suggests the final **";Bss=zz"** is optional. This is not the case. It must be specified.

4.7.9.5. Query About Supported Functionality

The **qSupported** packet asks the client for information about features for which support is optional. By default, none are supported. The features are maximum packet size and support for the various **qXfer** packets and the **QPassSignals** packet.

Of these only the packet size is of relevance to this target, so a reply of **"PacketSize=xx"**, where **"xx"** is the maximum packet size (**GDB_BUF_MAX**) is sent.

4.7.9.6. Query About Symbol Table Data

A **qSymbol::** packet (i.e. a **qSymbol** packet with no data) is used as an offer from the client to provide symbol table information. The server may respond with packets of the form **qSymbol:name** to request information about the symbol **name**.

A reply of **"OK"** is used to indicate that no further symbol table information is required. For the current implementation, no information is required, so **"OK"** is always sent as the response.

4.7.10. Set Packets

Set packets all start with **Q**. The functionality is all provided in **rsp_set ()**.

The **QPassSignals** packet is used to pass signals to the target process. This is not supported, and not reported as supported in a **qSupported** packet (see Section 4.7.9.5), so should never be received.

If a **QPassSignals** packet is received, an empty response is used to indicate no support.

4.7.10.1. Tracepoint Packets

All the remaining set packets (**QTDP**, **QFrame**, **QTStart**, **QTStop**, **QTinit** and **QTro**) are concerned with tracepoints. Tracepoints are not currently supported with the *Or1ksim* target, so an empty reply packet is sent to indicate this.

4.7.11. Restart the Target

The functionality for the **R** packet is provided in **rsp_restart ()**. The start address of the current target is held in **rsp.start_addr**. The program counter is set to this address using **set_npc ()** (see Section 4.6.5).

The processor is not unstalled, since there would be no way to regain control if this happened. It is up to the GDB client to restart execution (with **continue** or **step** if that is what is desired).

This packet should only be used in extended remote debugging.

4.7.12. Stepping

The step packet (**s**) requests a single machine instruction step. Its implementation is almost identical to that of the continue (**c**) packet, but using the functions **rsp_step ()** and **rsp_step_generic ()**.

The sole difference is that the generic function sets, rather than clears the single stepping flag in Debug Mode Register 1. This ensures a TRAP exception is raised after the next instruction completes execution.

```
cpu_state.sprs[SPR_DRR]   = 0;
cpu_state.sprs[SPR_DMR2] &= ~SPR_DMR2_WGB;
cpu_state.sprs[SPR_DMR1] |= SPR_DMR1_ST;
cpu_state.sprs[SPR_DSR]  |= SPR_DSR_TE;
```

4.7.13. v Packets

The **v** packets provide additional flexibility in controlling execution on the target. Much of this is related to non-stop targets with multithreading support and to flash memory control and need not be supported in a simple implementation.

All the **v** packet functionality is provided in the function **`rsp_vpkt ()`**.

4.7.13.1. Extended Debugging Support

The **vAttach** and **vRun** packets are only required for extended remote debugging.

vRun is used to specify a new program to be run, or if no program is specified that the existing target program be run again. In the current implementation, only this latter option is supported. Any program specified is ignored with a warning. The semantics of the **vRun** command are that the target is left in the stopped state, and the stopped condition reported back to the client.

The **vRun** packet may also specify arguments to pass to the program to be run. In the current implementation those arguments are ignored with a warning.

This behavior is identical to that of the **R** (restart) packet (see Section 4.7.11) with the addition of a reply packet. The implementation uses exactly this functionality, with a reply packet reporting a TRAP exception.

```
      rsp_restart ();
      put_str_packet ("S05");
```

The **vAttach** packet allows a client to attach to a new process. In this target, there is only one process, so the process argument is ignored and no action taken. However a stop response is required, so a reply packet indicating a TRAP exception is sent (**`put_str_packet ("S05")`**.

4.7.13.2. Non-stop Support

The **vCont** packet provides a more fine grained control over individual threads than the **c** or **s** packets.

Support for **vCont** packets is established with a **vCont?** packet which should always be supported. In the current implementation, **vCont** is not supported, so an empty response is provided to any **vCont?** packet.

4.7.13.3. File Handling

The **vFile** packet allows a file operation to be implemented on a target platform. In the absence of any file system with the "bare metal" target, this packet is not supported. An empty response is sent and a warning printed.

4.7.13.4. Flash Memory

The **vFlashErase**, **vFlashWrite** and **vFlashDone** packets provide support for targets with flash memory systems.

At present these are not supported on the target and an error reply ("E01") is returned. However *Or1ksim* can model flash memory, and these packets could be supported in the future.

4.7.14. Binary Data Transfer

The **X** provides for data to be written to the target in binary format. This is the mechanism of choice for program loading (the GDB **load** command). GDB will first probe the target with an empty **X** packet (which is 7-bit clean). If an **"OK"** response is received, subsequent transfers will use the **X** packet. Otherwise **M** packets will be used. Thus even 7-bit clean implementations should still support replying to an empty **X** packet.

The example implementation is found in **rsp_write_mem_bin ()**. Even though the data is binary, it must still be escaped so that '#', '$' and '}' characters are not mistaken for new packets or escaped characters.

Each byte is read, and if escaped, restored to its original value. The data is written using **set_program8 ()**, having first verified the memory location with **verify_memoryarea ()**.

If all bytes are successfully written, a reply packet of **"OK"** is sent. Otherwise and error packet ("E01") is sent. The error number does not matter—it is ignored by the target.

Caution

The use of **set_program8 ()** is not correct, since it ignores any caching or memory management. As a result the current implementation is only correct for configurations with no instruction MMU or instruction cache.

4.7.15. Matchpoint Handling

Matchpoint is the general term used for breakpoints (both memory and hardware) and watchpoints (write, read and access). Matchpoints are removed with **z**command> packets and set with **Z** packets. The functionality is provided respectively in **resp_remove_matchpoint ()** and **resp_insert_matchpoint ()**.

The current implementation only supports memory (soft) breakpoints controlled by **Z0** and **Z0** packets. However the OpenRISC 1000 architecture and *Or1ksim* have hardware breakpoint and watchpoint functionality within the debug unit, which will be supported in the future.

The target is responsible for keeping track of any memory breakpoints set. This is managed through the hash table pointed to by **rsp.mp_hash**. Each matchpoint is recorded in a matchpoint entry:

```
struct mp_entry
{
  enum mp_type       type;
  unsigned long int  addr;
  unsigned long int  instr;
  struct mp_entry   *next;
};
```

When an instruction is replaced by **l.trap** for a memory breakpoint, the replace instruction is recorded in the hash table as a **struct mp_entry** with type **BP_MEMORY**. This allows it to be replaced when the the breakpoint is cleared.

The hash table is accessed by the functions **mp_hash_init ()**, **mp_hash_add ()**, **mp_hash_lookup ()** and **mp_hash_delete ()**. These are described in more detail in Section 4.3.2.1

4.7.15.1. Setting Matchpoints

Only memory (soft) breakpoints are supported. The instruction is read from memory at the location of the breakpoint and stored in the hash table (using **mp_hash_add ()**). A **l.trap** instruction (**OR1K_TRAP_INSTR**) is inserted in its place using **set_program32 ()** and a reply of **"OK"** sent back.

```
mp_hash_add (type, addr, eval_direct32 (addr, 0, 0));
set_program32 (addr, OR1K_TRAP_INSTR);
put_str_packet ("OK");
```

Caution

The use of **eval_direct32 ()** with second and third arguments both zero and **set_program32 ()** is not correct, since it ignores any caching or memory management. As a result the current implementation is only correct for configurations with no instruction MMU or instruction cache.

4.7.15.2. Clearing Matchpoints

Only memory (soft) breakpoints are supported. The instruction that was substituted by **l.trap** is retrieved and deleted from the hash table using **mp_hash_delete ()**. The instruction is then put back in its original location using **set_program32 ()**.

mp_hash_delete () returns the **struct mp_entry** that was removed from the hash table. Once the instruction information has been retrieved, its memory must be returned by calling **free ()**.

It is possible to receive multiple requests to delete a breakpoint if the serial connection is poor (due to retransmissions). By checking that the entry is in the hash table, actual deletion of the breakpoint and restoration of the instruction happens at most once.

```
mpe = mp_hash_delete (type, addr);

if (NULL != mpe)
  {
    set_program32 (addr, mpe->instr);
    free (mpe);
  }

put_str_packet ("OK");
```

Caution

The use of **set_program32 ()** is not correct, since it ignores any caching or memory management. As a result the current implementation is only correct for configurations with no instruction MMU or instruction cache.

Chapter 5. Summary

This application note has described in detail the steps required to implement a RSP server for a new architecture. That process has been illustrated using the port for the OpenRISC 1000 architecture.

Suggestions for corrections or improvements are welcomed. Please contact the author at **jeremy.bennett@embecosm.com**.

Glossary

big endian

A description of the relationship between byte and word addressing on a computer architecture. In a big endian architecture, the least significant byte in a data word resides at the highest byte address (of the bytes in the word) in memory.
The alternative is little endian addressing.

See also: little endian.

General Purpose Register (GPR)

In the OpenRISC 1000 architecture, one of between 16 and 32 general purpose integer registers.
Although these registers are general purpose, some have specific roles defined by the architecture and the ABI. GPR 0 is always 0 and should not be written to. GPR 1 is the stack pointer, GPR 2 the frame pointer and GPR 9 the return address set by **`l.jal`** (known as the link register) and **`l.jalr`** instructions. GPR 3 through GPR 8 are used to pass arguments to functions, with scalar results returned in GPR 11.

See also: Application Binary Interface.

Joint Test Action Group (JTAG)

JTAG is the usual name used for the IEEE 1149.1 standard entitled *Standard Test Access Port and Boundary-Scan Architecture* for test access ports used for testing printed circuit boards and chips using boundary scan.
This standard allows external reading of state within the board or chip. It is thus a natural mechanism for debuggers to connect to embedded systems.

little endian

A description of the relationship between byte and word addressing on a computer architecture. In a little endian architecture, the least significant byte in a data word resides at the lowest byte address (of the bytes in the word) in memory.
The alternative is big endian addressing.

See also: big endian.

Memory Management Unit (MMU)

A hardware component which maps virtual address references to physical memory addresses via a page lookup table. An exception handler may be required to bring non-existent memory pages into physical memory from backing storage when accessed.
On a Harvard architecture (i.e. with separate logical instruction and data address spaces), two MMUs are typically needed.

Real Time Executive for Multiprocessor Systems (RTEMS)

An operating system for real-time embedded systems offering a POSIX interface. It offers no concept of processes or memory management.

Special Purpose Register (SPR)

In the OpenRISC 1000 architecture, one of up to 65536 registers controlling all aspects of the processor. The registers are arranged in groups of 2048 registers. The present architecture defines 12 groups in total.

In general each group controls one component of the processor. Thus there is a group to control the DMMU, the IMMU, the data and instruction caches and the debug unit. Group 0 is the system group and includes all the system configuration registers, the next and previous program counters, supervision register and saved exception registers.

System on Chip (SoC)

A silicon chip which includes one or more processor cores.

References

[1] Embecosm Application Note 2. The OpenCores OpenRISC 1000 Simulator and Tool Chain: Installation Guide. Embecosm Limited, June 2008.

[2] Embecosm Application Note 3. Howto: Porting the GNU Debugger: Practical Experience with the OpenRISC 1000 Architecture Embecosm Limited, August 2008.

[3] Debugging with GDB: The GNU Source-Level Debugger, Richard Stallman, Roland Pesch, Stan Shebbs, et al, issue 9. Free Software Foundation 2008 . http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

[4] GDB Internals: A guide to the internals of the GNU debugger, John Gillmore and Stan Shebbs, issue 2. Cygnus Solutions 2006 . http://sourceware.org/gdb/current/onlinedocs/gdbint_toc.html

[5] Doxygen: Source code documentation generator tool, Dimitri van Heesch, 2008 . http://www.doxygen.org

[6] OpenRISC 1000 Architectural Manual, Damjan Lampret, Chen-Min Chen, Marko Mlinar, Johan Rydberg, Matan Ziv-Av, Chris Ziomkowski, Greg McGary, Bob Gardner, Rohit Mathur and Maria Bolado, November 2005 . http://www.opencores.org/cvsget.cgi/or1k/docs/openrisc_arch.pdf

[7] OpenRISC 1000: ORPSoC Damjan Lampret et al. OpenCores http://opencores.org/projects.cgi/web/or1k/orpsoc

[8] SoC Debug Interface Igor Mohor, issue 3.0. OpenCores 14 April, 2004 . http://opencores.org/cvsweb.shtml/dbg_interface/doc/DbgSupp.pdf

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

