Canon Research Centre France S.A

Rue de la touche Lambert

35515 CESSON SEVIGNE Cedex
FRANCE

Project : TOOLS	Document title :	RTEMS Remote Debugger Server specs	
Created on : 29/09/98	**Last Modified on :** 05/10/98		**Version :** 1.0

Author : Emmanuel RAGUET, Eric VALETTE   CRF   TA Dept.

Document Internal Reference :   98.TA.0373.M.ER

Verified By :

Name : E. Valette	**Position :** Tools project Leader	
Company : CRF	**Department :** TA	**Sign :**
Received on :	**Deadline :**	

Validated By :

Name : E. Valette	**Position :** Tools project Leader	
Company : CRF	**Department :** TA	**Sign :**
Received on :	**Deadline :**	

DISTRIBUTION LIST

Name	Company	Department	Position	Reason	Has Copy
H. Dias	CRF	TA	Manager	Info	1 (Paper)
E. Valette	CRF	TA	Project Leader	Approval	2 (Paper)
E. Raguet	CRF	TA	Engineer	Author	3 (Paper)

RTEMS Remote Debugger

Contents

List of Figures

1   Introduction

The TOOLS project aims to provide a complete development environment for RTEMS OS. This environment must be as close as possible to the Chorus one (gnu compiler, gnu linker, gnu debugger, ...), because it is currently the OS which is the most used at CRF and we want to simplify the migration path from the ChorusOs environment to the RTEMS environment. One important item of this development environment is the remote debugger which allows the developer to debug his software on a target machine from a host machine via a communication link (Ethernet, serial link, ...).

The choice of GDB as debugger has been made with because in CRF, every developer, which uses the ChorusOs development environment, debugs his software using the remote debugging functionality of GDB. Providing a remote GDB debug server running on RTEMS, will enable the developers to use transparently the same debugger for a different RTOS. Another reason for the choice of GDB is its constant evolution, and that it can be interfaced with graphic user interfaces like DDD providing a very powerfull debugging environment.

The subject of this document is to explain how GDB works and the way to implement a daemon on RTEMS that will be used as a debugger server for a GDB client. We will call this daemon Rtems GDB Debug Server Daemon (RGDBSD). We aim to provide this debugger running at least on 2 host systems : Linux 2.x and Solaris 2.5.1 as they are the two platforms used for developing Chorus applications today.

2   Document Revision History

Current release :

- Current applicable release is 1.1.

Existing releases :

- 0.1 : Released the 29/09/98. First draft of this document.

- 0.2 : Released the 05/10/98. Second draft version.

- 1.0 : Released the 08/10/98. Version Approved internally.

- 1.1 : Released the 13/13/98. Version Distributed for comments.

Planned releases :

- 2.0 depending on comments we get from the RTEMS development mailing list.

3   Objectives

This section is intended to clearly define the current objectives of our work. First, we will try here to list some ambitious requirements for the debugger in section 3.1. These requirements will deliberately be much more ambitious than what we will provide directly ourselves in the hope that the Internet development model will enable others to implement some features we rejected for man-power reasons in the first step. We are committed to do the core work and redistribute it but would appreciate any comment and enhancement. Then, in section 3.2 we will analyze each requirement to see what technical problem must be solved if we want to fullfill it. After this analysis, we will determine in section 3.3 the requirements we chose to implement and the ones we will not. We will then clearly identify the limits of our solution in section 3.4.

3.1   List of Requirements

We will identify here possible requirements for the type of debug that may be provided :

(R1) : We want to use GDB as the front-end debugger,

(R2) : We want to support at least Intel and PowerPC as target processor architecture,

(R3) : We want to use the GDB thread debugging interface,

(R4) : We want to be able to debug a remote target over a serial line,

(R5) : We want to be able to debug a remote target over Ethernet,

(R6) : The set of target code path we will be able to debug using RGDBSD must be clearly identified. It will be called Debug Path Set (*DPS*) in the remaining of this document,

(R7) : *DPS* must include the RTEMS core executive itself,

(R8) : *DPS* must include the FreeBSD stack,

(R9) : *DPS* must include anything but the FreeBSD stack and the RTEMS core executive,

(R10) : We want to enable several persons to debug different parts of the code running on the target,

(R11) : As much as possible the system must be frozen during a debug session so that debugging a particular portion of code does not prevent another part from functioning,

3.2   Requirements Analysis

(R1) : Worth recalling it. It mainly imposes few restrictions on the binary files type, target processor type as :

- the binary format must be understood by GDB (to find debugging information). Elf, Coff and A.out are the main formats currently supported. Elf/Dwarf 2.0 binary support will be our main target as they are the preferred format for Intel and PowerPC processors. No change in GDB will be required for other binaries except may be a new configuration file changing the binary/debug file format,
- the processor must be supported for disassemble/step instruction command,
- the target system must be supported. As far as I know RTEMS is not currently *officially* supported anyway,

(R2) : Our primary targets are Intel and PowerPC. We however do not think implementing RGDBSD for other processors will be a heavy task. It will mainly require :

1. Implementing exception handling for the target processor,
2. Interfacing the generic part of RGDBSD with the low level exception handling and make RGDBSD aware of exception used for debugging (usually illegal instruction or dedicated trap, single step),
3. Making GDB aware of the frame layout pushed on exceptions,
4. Implement the code for data transfer for the exception frame,

As soon as we will have completed the first core work a document describing how to port it to a new processor should be written.

(R3) : GDB already has an interface for manipulating multi-threaded programs. This interface is rather weak at the moment but it will certainly be improved in the future with the generalization of POSIX thread API on Linux and other operating systems. This will place some requirements on the RTEMS API (something like a **threadList** system call),

(R4) : Regular GDB code contains clients code for debugging over a serial line. However only few functions are implemented. We would like to provide a better support and to uniformize serial line debugging with debugging over Ethernet via the use of SLIP,

(R5) : Regular GDB code contains client code for debugging over Ethernet for VxWorks via the SUN RPC library. So there will be at least one starting point to implement remote debugging over Ethernet via SUN RPC. The Chorus remote debugging code should be disclosed under GPL in the coming weeks,

(R6) : Due to a classical chicken and egg problems, the remote debugging daemon cannot be used to debug code it uses to function. Thus depending on the API used by RGDBSD, some parts of the target system will not be debuggable via GDB. The most important point is documentation because my feeling is that implementing RGDBSD on a totally different *dedicated* nano kernel should be possible,

(R7) : RTEMS core executive is a real-time OS which implements priority level scheduling, synchronization objects, and interrupt handling. As mentioned in previous item, we may not debug theses features if RGDBSD uses them. This requirement is thus very strong because it impose that :

1. RGDBSD is totally interrupt driven (no thread API available),

2. But it does not use RTEMS interrupt management,

3. Nor does not use RTEMS exception management,

4. RGDBSD must provide its own UDP/IP stack as the current FreeBSD code rely on tasks switching and RTEMS provided synchronization object for input path handling,

So our feeling is that the **(R7)** more or less requires to write a *dedicated* nano kernel with a very small dedicated UDP/IP stack.

(R8) : GDB remote debugging over Ethernet code communicates with the remote target via the SUN RPC protocol. This requires a UDP/IP protocol and a minimal socket like interface. In RTEMS environment, this feature is currently provided by the FreeBSD stack. Again, if we use the FreeBSD stack itself for remote communication, it will be impossible to debug this stack as a breakpoint in the stack code will stop its execution and there would be no more way to communicate with the target. A solution consists in implementing a minimal, dedicated UDP/IP stack (with at least IP and UDP protocols, a minimal BSD sockets) and a simple SUN RPC library, which will be both dedicated to the debug. We can use RTEMS API to implement it if **(R7)** is not required. As the two stack will need to share the same chip, a kind of shared filter must be implemented at the bottom of the two stacks so that Ethernet frames can be dynamically directed either to the dedicated UDP/IP debug stack or to the regular FreeBSD stack. The fact that in the current design, the low level ethernet input routine mainly signal a thread should facilitate the design of this filter. The output path is less complicated as it is run by a task and thus can sleep on a synchronization object,

(R9) : This requirement represents what we find reasonable as a first target. However, we can still present to the final user this kind of debugging via different model. RTEMS can be represented as a single threaded system or, because RTEMS is a multitasking system, as an ensemble of separate tasks. In the first representation, the debugger sees only 1 "task" without distinguishing the core executive part from the applicative part. This is the simplest way to implement the debugger but also implies that there is no way to protect the core

executive. Some of these tasks are system tasks (tasks form the core executive and from the FreeBSD stack), the other ones are tasks implemented by the developer. The developer wants to debug his tasks, and sometimes only one of his tasks. We can provide a way to debug not the entire system but only the concerned task by testing if the current running task is a debugged task (test on the task identifier). GDB offers an API to "detach" thread so that if a detached thread hits a breakpoint it is automatically restarted without user intervention,

(R10) : Several developers can work on a large project, each on a specific module. Sometimes only one target is available for everyone. This requirements is not really meaningfull until RTEMS supports dynamic code loading,

(R11) : This requirement heavily depends on the **(R7)** and **(R8)** requirements.

3.3   Requirements Selection

3.3.1   Requirement We Will Take Into Account For the First Implementation

(R1) : Obviously.

(R2) : As these are our targets. Of course other will be free to contribute. We will however document the works that needs to be done in order to port the debug code to other processors,

(R3) : We think it is feasible with only few RTEMS modifications,

(R5) : We think serial line debugging is nowadays too restrictive as most equipment are now connected via Ethernet,

(R6) : This is a documentation problem and should be fairly easy to describe once we have the RGDBSD code,

(R9) : We will try to provide the multi-thread target system presentation,

3.3.2   Requirements We Will Not Implement

(R4) : it will not be implemented for the moment. It is just a matter on implementing SLIP in the FreeBSD stack and alternative solutions already exist in the meantime,

(R7) : To simplify the first developments, we don't plan to implement a *dedicated* nano-kernel to allow the RTEMS kernel to be debugged. It means that, if any breakpoint is set in the kernel, unpredictable behaviors can occur. So, developers must keep in mind to avoid stopping the kernel. They must also keep in mind, in order to not stop the kernel, that the user's tasks must have a lower priority than the tasks used for debug. The solution is to use a specific very-high priority level for the system tasks used directly or indirectly by RGDBSD. The SYSTEM_TASK attribute that already exists should be fine.

(R8) : To avoid increasing the code size and the used memory and because the FreeBSD stack doesn't need to be debug any more, we choose not to implement a minimal TCP/IP stack but rather to use the FreeBSD one as communication protocol,

(R10) : We will see later when a file system will be available and we can implement **exec** system call,

(R11) : Without a separate TCP/IP stack it will be hard to freeze the system as some interrupts must occur in order to enable the FreeBSD stack to function,

3.4 Implied Restrictions

High priority level must be used for these features :

- FreeBSD interrupt handling thread,

- Debugger threads.

This will allows these tasks not to be stopped when a process is stopped in debug mode
If we don't want to use a "specific" priority level, we must affect priority to each of these tasks as follow :

- FreeBSD stack (high priority)

- Debugger (less high priority)

The user must remember the higher priority level he can use for his software in order not to block one of the previous threads and to not put breakpoints in part of the code executed by RGDBSD.

4 A Rapid Tour of GDB Internals

To help the reader to understand what needs to be implemented, we will present briefly how GDB works regardless if the target is local or remote. A debugger is a tool which enables control of the execution of software on a target system. In most of cases, the debugger connects to a target system, attaches a process, inserts breakpoints and resumes execution. Then the normal execution is completely events driven (process execution stopped due to a breakpoint, process fault, single-step,...) coming from the debuggee. It can also directly access some parts of the target processor context (registers, data memory, code memory,...) and change their content. Native GDB debugger can just be seen as special cases where the host and the target are on the same machine and GDB can directly access the target system debug API.

In our case, the host and the target are not on the same machine and an Ethernet link is used to communicate between the different machines. Because GDB needs to be able to support various targets (including Unix core file, ...), each action that needs to be performed on the debuggee is materialized by a field of the following *targets_op*s structure :

```
struct target_ops
{
  char          *to_shortname;   /* Name this target type */
  char          *to_longname;    /* Name for printing */
  char          *to_doc;         /* Documentation.  Does not include trailing
                                    newline, and starts with a one-line descrip-
                                    tion (probably similar to to_longname). */
  void          (*to_open) PARAMS ((char *, int));
  void          (*to_close) PARAMS ((int));
  void          (*to_attach) PARAMS ((char *, int));
  void          (*to_detach) PARAMS ((char *, int));
  void          (*to_resume) PARAMS ((int, int, enum target_signal));
  int           (*to_wait) PARAMS ((int, struct target_waitstatus *));
  void          (*to_fetch_registers) PARAMS ((int));
  void          (*to_store_registers) PARAMS ((int));
  void          (*to_prepare_to_store) PARAMS ((void));

  /* Transfer LEN bytes of memory between GDB address MYADDR and
     target address MEMADDR.  If WRITE, transfer them to the target, else
```

```
        transfer them from the target.  TARGET is the target from which we
        get this function.

        Return value, N, is one of the following:

        0 means that we can't handle this.  If errno has been set, it is the
        error which prevented us from doing it (FIXME: What about bfd_error?).

        positive (call it N) means that we have transferred N bytes
        starting at MEMADDR.  We might be able to handle more bytes
        beyond this length, but no promises.

        negative (call its absolute value N) means that we cannot
        transfer right at MEMADDR, but we could transfer at least
        something at MEMADDR + N.   */

    int         (*to_xfer_memory) PARAMS ((CORE_ADDR memaddr, char *myaddr,
                                           int len, int write,
                                           struct target_ops * target));

    void        (*to_files_info) PARAMS ((struct target_ops *));
    int         (*to_insert_breakpoint) PARAMS ((CORE_ADDR, char *));
    int         (*to_remove_breakpoint) PARAMS ((CORE_ADDR, char *));
    void        (*to_terminal_init) PARAMS ((void));
    void        (*to_terminal_inferior) PARAMS ((void));
    void        (*to_terminal_ours_for_output) PARAMS ((void));
    void        (*to_terminal_ours) PARAMS ((void));
    void        (*to_terminal_info) PARAMS ((char *, int));
    void        (*to_kill) PARAMS ((void));
    void        (*to_load) PARAMS ((char *, int));
    int         (*to_lookup_symbol) PARAMS ((char *, CORE_ADDR *));
    void        (*to_create_inferior) PARAMS ((char *, char *, char **));
    void        (*to_mourn_inferior) PARAMS ((void));
    int         (*to_can_run) PARAMS ((void));
    void        (*to_notice_signals) PARAMS ((int pid));
    int         (*to_thread_alive) PARAMS ((int pid));
    void        (*to_stop) PARAMS ((void));
    enum strata   to_stratum;
    struct target_ops
                *DONT_USE;      /* formerly to_next */
    int         to_has_all_memory;
    int         to_has_memory;
    int         to_has_stack;
    int         to_has_registers;
    int         to_has_execution;
    struct section_table
                *to_sections;
    struct section_table
                *to_sections_end;
    int         to_magic;
    /* Need sub-structure for target machine related rather than comm related? */
};
```

This structure contains pointers to functions (in C++, this would be called a virtual class). Each

different target supported by GDB has its own structure with the relevant implementation of the functions (some functions may be not implemented). When a user connects GDB to a target via the "target" command, GDB points to the structure corresponding to this target. Then the user can attache GDB to a specific task via the "attach" command. We have therefore identified two steps to begin a remote debug session :

1. the choice of the target type (in our case RTEMS),

2. the choice of what to debug (entire system, specific task,...),

Note that in the case of natives debugger, the choice of the target is implicitly performed by commands like **run**, **attach**, **detach**. Several figures will now be described showing the main steps of a debug session.

Figure 1 explains how the debugger connects to the target :

1. The debugger opens a connection to the target. The word "connection" doesn't only mean Ethernet or serial link connection but all the ways by which a process can communicate with another one (direct function call, messages mailbox, ...),

2. The targets checks if it can accept or reject this connection,

3. If the connection is accepted, the host "attaches" the process,

4. the target stops the process, notifies a child's stop to the host and waits for command,

5. the host can ask information about the debugged process (name, registers,...) or perform some action like setting breakpoints, ...

Figure 2 explains how the debugger manages the breakpoints and controls the execution of a process :

1. The host asks the debuggee what is the opcode at the concerned address in order for GDB to memorize this instruction,

2. the host sends a CONTINUE command : it asks the target to write the "DEBUG" opcode (for example, the INTEL "DEBUG" opcode is INT3 which generate a breakpoint trap) instead of the debugged opcode.

3. then the host waits for events,

4. after the change of instruction, the target resumes the execution of the debuggee,

5. when the "DEBUG" opcode is executed, the breakpoint exception handler is executed and it notifies the host that the process is stopped. Then it waits for commands (if no command is sent after a certain amount of time, the connection will be closed by the target).

6. the host asks the target to re-write the right opcode instead of the "DEBUG" opcode and then can ask information

Figure 2 also shows the case of other "CONTINUE" commands (remember that the "DEBUG" opcode has been replaced by the right instruction):

1. Host sends first a "single step" command to execute the debugged instruction,

2. It then waits for "single step" exception event,

3. the target, once the single step executed, calls the debug exception handler. It notifies the host that execution is suspended and wait for commands.

4. the host asks the target to re-write the "DEBUG" opcode (breakpoint trap) instead of the debugged one.

5. then the host sends a "CONTINUE" command in order the target to resume the process execution to the next breakpoint.

Figure 3 explains how the debugger disconnects from a target :

1. the host sends a detach command to the target.

2. the target detaches the concerned process, notifies the detachment and resumes the process execution.

3. once notified, the host sends a close connection command.

4. the target closes the connection.

These 3 examples show that the mains actions that are performed by the host debugger on the target are only simple actions which look like :

- read/write code,

- read/write data,

- read/write registers,

- manage exceptions,

- send/receive messages to/from the host.

5   Interfacing GDB with RTEMS as a Target

So, basically, porting GDB to RTEMS environment requires implementing the functions contained in the target_ops structure. The native debugger implementation (where the host machine is also the target one) uses direct function calls. For our needs (remote debugging), these functions must be implemented to support the encapsulation in UDP/IP layers and communications between different types of host machines : the best solution is use the SUN Remote Procedure Calls API (SUN RPC). This SUN RPC module will be explained below (see paragraph 6).
We can note that the functions described in the target_ops structure are high-level functions. The main reason is that GDB was designed in order to be able to use monitor firmware as a debug server. In the case of a Unix OS target, these high-level functions are implemented themselves using a lower level POSIX API. Because we want to simplify the code running on the target and decrease its size of this code, we propose to use the POSIX layer API used for the debug like **waitpid**, **ptrace**, ... Due to the GDB working mode and due to our requirements, we can establish here a non-exhaustive list of some commands required to implement the previously described functions :

- set up a connection with a target,

- close a connection,

- send a signal to the specified process,

- get a list of process/thread/connection running,

- control process under debug,

- ...

TARGET INDEPENDENT PART

TARGET DEPENDENT PART

messages exchange

Connect on target

Open connection

Accept connection

Attach a process

Attach a process

Stop the process
Notify a child change

Ask information

Give information

...

Figure 1: Debug session initialization

TARGET
INDEPENDENT
PART

message exchange

TARGET
DEPENDENT
PART

Set Breakpoint

Ask opcode at address

...

Send opcode

Continue process : first time

Write "DEBUG" opcode
at address

Resume process
Wait for event...

DEBUG opcode
Breakpoint Except. handler
Notify a child change
Wait for command ...

Write previous opcode
at address

Ask information

...

Give information

Continue process : next times

Single Step command
Wait for event...

Debug Exception handler
Notify a child change
Wait for command ...

Write "DEBUG" opcode
at address

Continue command
Wait for event...

Resume process
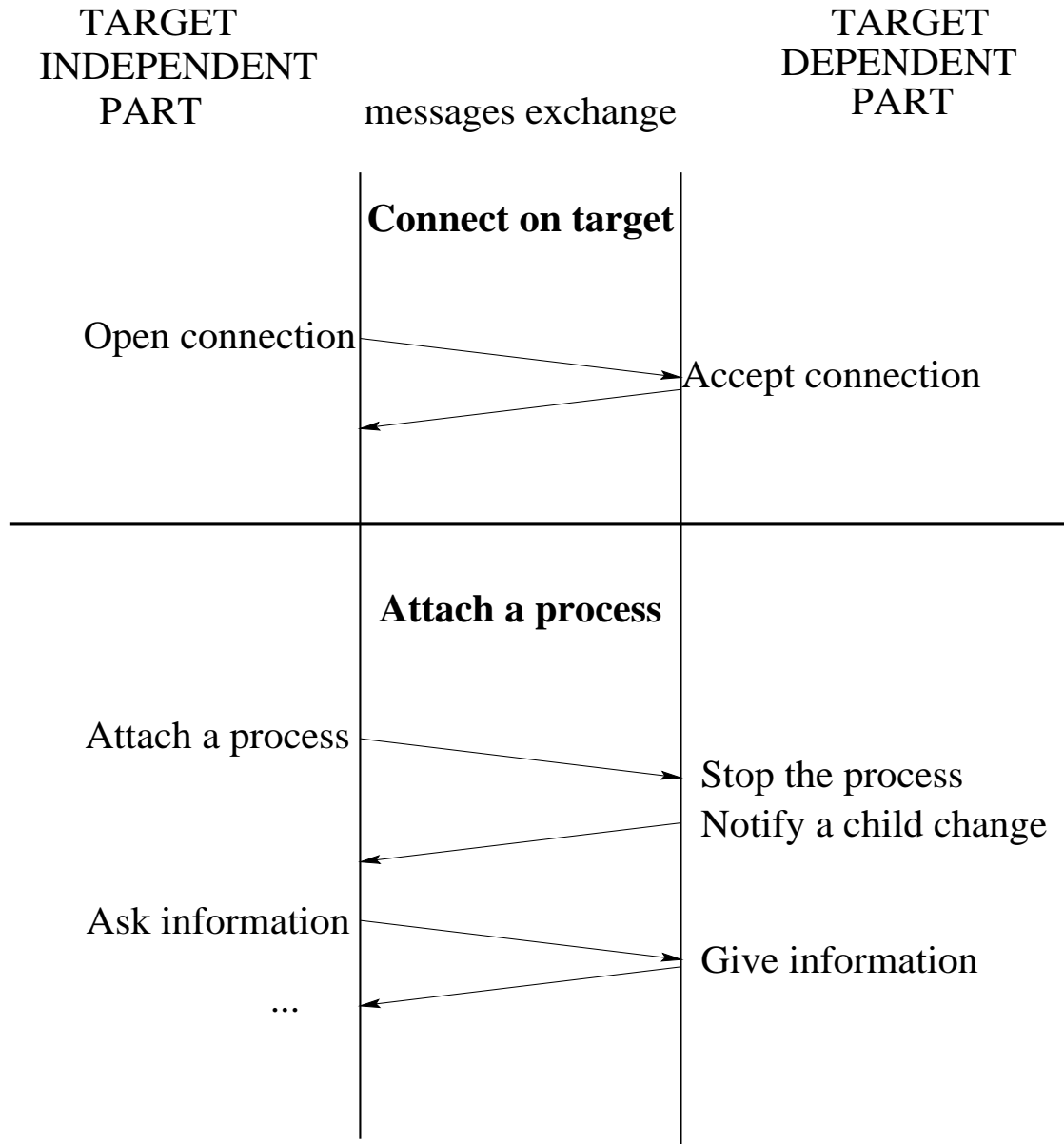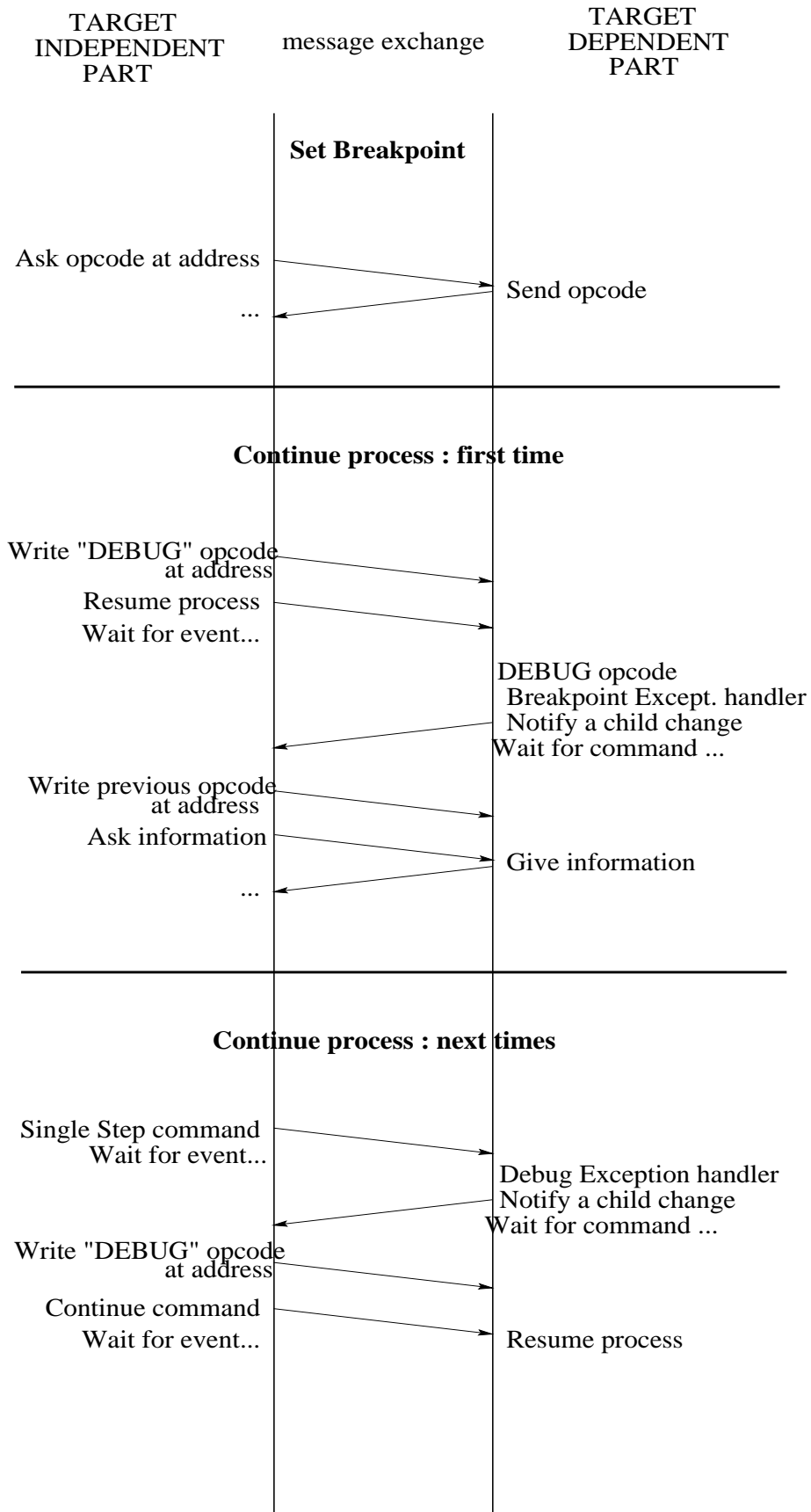
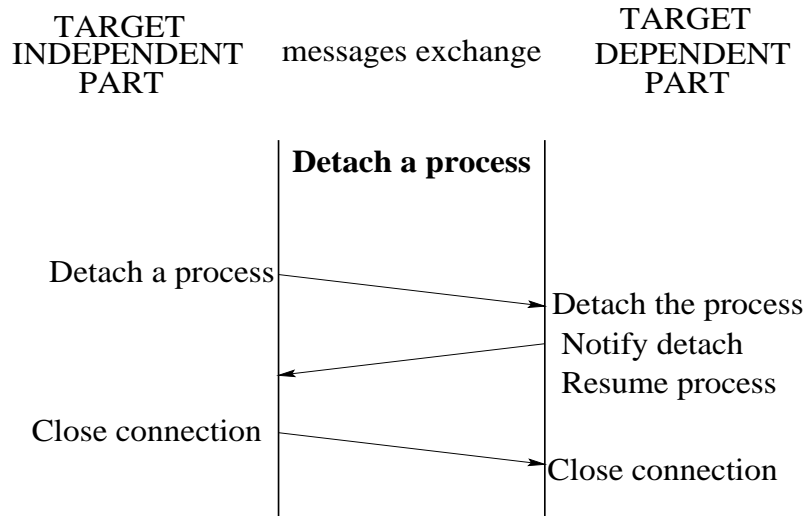Figure 2: Breakpoint and process execution

Figure 3: Detach a process and close a connection

Control means that, with this function, we can read, write the memory of the debuggee, insert breakpoint to stop the process and resume the process execution. This command can be implemented by emulating in the RTEMS environment a Unix function called "ptrace". This function allows the control of a child process. The "ptrace" function has some sub-functions which are described below (some of these actions and standardized, the others are added due to our needs) :

- PTRACE_PEEKTEXT, PTRACE_PEEKDATA : read word at address,

- PTRACE_POKETEXT, PTRACE_POKEDATA :write word at address,

- PTRACE_CONT : restart after signal,

- PTRACE_KILL : send the child a SIGKILL to make it exit,

- PTRACE_SINGLESTEP : set the trap flag for single stepping,

- PTRACE_ATTACH : attach to the process specified,

- PTRACE_DETACH : detach a process that was previously attached.

This list only contains the command that are described in the ptrace Unix manpage. For some specific needs (debug of one task among several ones, register read/write,...), it is possible to create some special ptrace commands as described after :

- get current task registers,

- set current task registers,

- list of the threads,

- identifier of the target thread,

- restart to address,

- set breakpoint at address,

- clear breakpoint,

- get breakpoints,

- load dynamically a task,

- ...

This list is not exhaustive and can be increased due to the needs. All the described functions will not be implemented in a first version, only the strictly needed. If some commands are added, the modifications must be implemented both in RTEMS and in GDB.

6 Communication with GDB

The RTEMS remote debugger will be accessed by GDB on a host machine through a communication link. We will use the TCP/IP stack included in RTEMS : the FreeBSD stack. The communication link will be based based on the UDP protocol and the BSD sockets which are parts of the FreeBSD stack. On top of these layers, we will plug a module which allows a simple communication between different machines (especially between different endianess machines) : the SUN Remote Procedure Call (SUN RPC). This code is freely available on the net and comes with a BSD like license. With this module, a process can invoke a procedure on a remote system. The RTEMS remote debugger will be seen by GDB as a SUN RPC server. Commands will be packed by the GDB SUN RPC client and sent to the server. This server will unpack these commands, execute them and, if needed, return results to the SUN RPC client.

Only a minimal subset of the SUN RPC library must be implemented. For example, the portmapper related API which allows a dynamic allocation of port numbers will not be implemented and some specific UDP port numbers will be used to establish the communication between the host and the target. The SUN RPC library implements the XDR module (eXternal Data Representation) which is a standard way of encoding data in a portable fashion between different endian systems. Below are figures describing the additional code and data size for the minimal library implementation we currently have already implemented for RTEMS :

```
size -x librpc.a
text  data bss dec hex filename
0x40e 0x0 0x0 1038 40e rpc_callmsg.o (ex librpc.a)
0x2f1 0x18 0x0 777 309 rpc_prot.o (ex librpc.a)
0x458 0x0 0x0 1112 458 svc.o (ex librpc.a)
0x4f 0x4 0x0 83 53 svc_auth.o (ex librpc.a)
0x75c 0x18 0x0 1908 774 svc_udp.o (ex librpc.a)
0x711 0x4 0x10 1829 725 xdr.o (ex librpc.a
0x149 0x0 0x0 329 149 xdr_array.o (ex librpc.a)
0x165 0x20 0x0 389 185 xdr_mem.o (ex librpc.a)
```

We have a constraint with the use of the UDP protocol. Because this protocol is connectionless, it is impossible, especially for the target, to detect if the connection is always active. On the other hand, using the TCP/IP protocols seems to be heavy especially if we plan to implement a dedicated micro stack for debug in the future. It can be a real problem to let the debugged process stopped during a long time even if there is no more debugger connected to the system. To avoid such a problem, the target must periodically test the connection with the host on another way than the one used to receive the commands. We must therefore open two communication ways so we need two fixed UDP port numbers.

1. One port will be used by the debugger to send its commands to the debugged process and to receive the result of these commands. View from the remote debugger, this port will be called primary port. For this one, we choose arbitrarily the port number 2000.

2. The other socket will be used as secondary port by the target to sometimes test the connection between the host and the target. These tests will occur in specific situations, when a process will be stopped on a breakpoint, single step instruction or other means. This secondary port will also be used by the target to signal any change in the behavior of a debugged process (stopped, killed, waiting for,...). For the secondary port, we choose the port number 2010.

These two port numbers are used by the remote debugger to open the two communication sockets. GDB will use its own mean to choose its port numbers (probably the Unix portmapper). The figure 4 shows the different layers we need to implement.

7   RTEMS Debugger Server Daemon

We will describe in this section how this debugger server will be implemented on RTEMS environment. Our initial target is based on Intel Pentium and we will use an Ethernet link to communicate between the host and the target.
The RTEMS remote debugger will be composed by several tasks and exception handlers :

- an initialization task which opens the sockets and runs the SUN RPC server. This task will also connect the interrupt handlers and launch the communication task

- a communication task which receives the SUN RPC commands, executes them and sends the result to the GDB client,

- a debug exception handler which manages the hardware breakpoint and single step exceptions (INT 1 on Intel x86),

- a breakpoint exception handler which manages the software breakpoints exceptions (INT 3 on Intel x86).

Figure 5 represents these different tasks and handlers.

7.1   The INITIALIZATION task

This is the task that must be executed at the boot phase of RTEMS. It initializes the debug context. It must :

- open the UDP sockets,

- run the SUN RPC server main loop,

- create the COMMAND MANAGEMENT task,

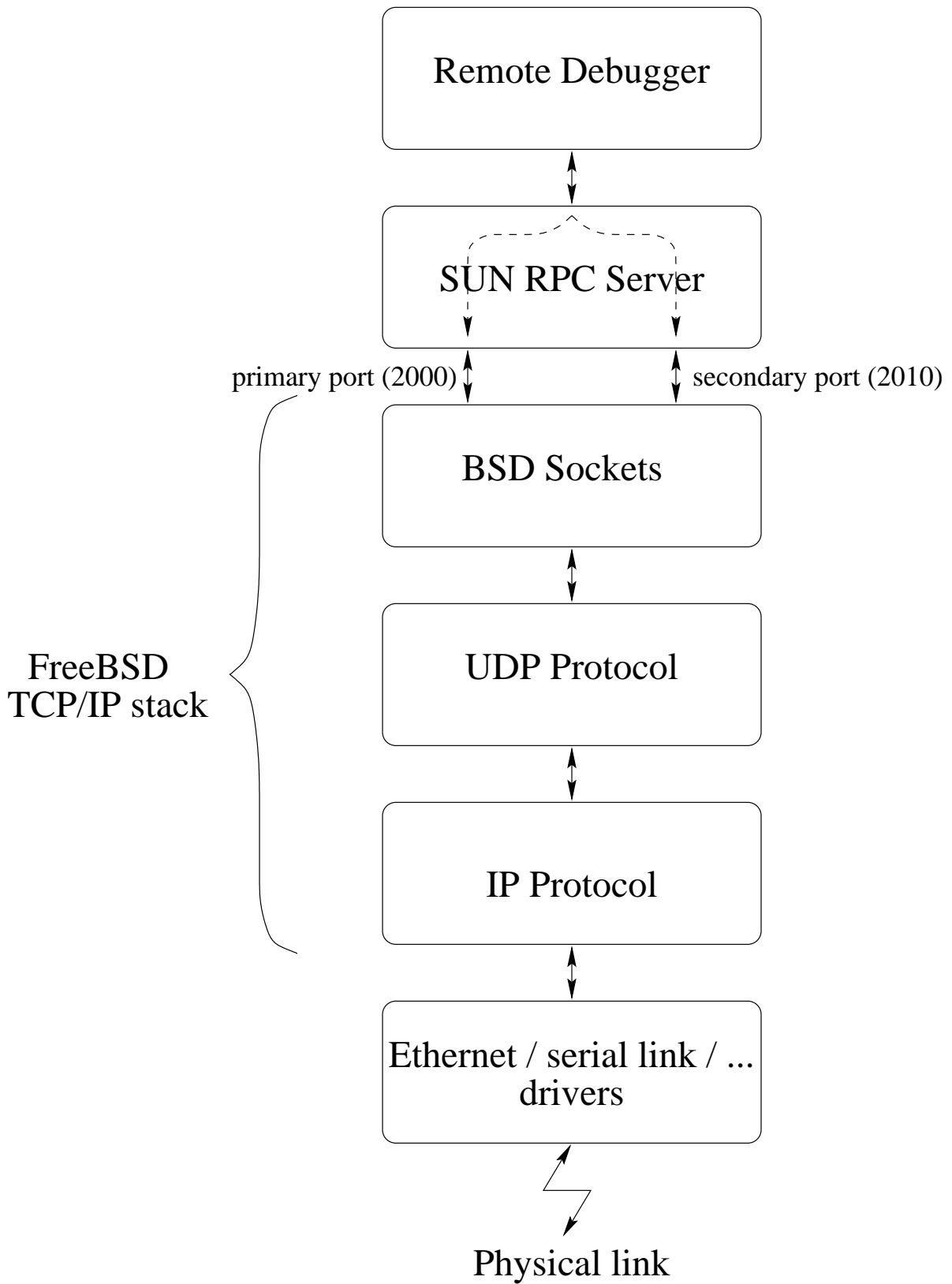- connect the DEBUG EXCEPTION handler,

15

Figure 4: Communication layers

- connect the SOFTWARE BREAKPOINT handler,

- delete itself.

If an error occurs at any step of the execution, the connections established before the error will be closed, before the initialization task deletes itself.

7.2 The COMMAND MANAGEMENT task

This task is in charge of receiving the SUN RPC messages and executing the associated commands. This task must have an important priority because it must be executed each time a command message comes from the debugger. It must be executed even if one or both exception handlers are executed. But the COMMAND MANAGEMENT task must not block the TCP/IP module without which no message can be received.
When not executing a command, this task is waiting for a SUN RPC message on the primary port. This idle state blocks the task, so the other active tasks can run. Once a message comes from Ethernet via the primary port, the COMMAND MANAGEMENT task wakes up and receives the message which is a request from GDB. This request is sent to the SUN RPC server which extracts the command and its arguments, executes it and, if needed, sends a result to GDB. After having performed these actions, the task sleeps, waiting for another message.

7.3 The DEBUG EXCEPTION handler

This handler is connected to the DEBUG exception (INT 1 on Intel ix86). This exception is entered when :

- executing a single-step instruction,

- hardware breakpoint condition is true,

These events will be treated by the debugger because they are the primary event used when debugging a software for instruction stepping. In both cases, the DEBUG EXCEPTION handler code is executed. Please note that the execution context of the exception handler is the supervisor stack of the task that generated the exception . This implies :

- We may sleep in this context,

- We have as many possible execution context for the DEBUG EXCEPTION handler as we need to,

First the exception handler sends a message to GDB via the secondary port to notify a process change : the debugged task is stopped. Then it will cause the faulting thread to sleep on a synchronization object. As soon as GDB receives the event notifying that the debuggee status has changed, it will start sending requests to get the debuggee status (registers set, faulty task id, ...). These requests are handled by the COMMAND MANAGEMENT task. When this task receive a PTRACE_CONT command it will resume the execution of the task that caused the exception by doing a V on the synchronization object.

7.4 The BREAKPOINT EXCEPTION handler

This handler is connected to the BREAKPOINT exception (INT3 on Intel Ix86). Each time the debugger wants to place a software breakpoint in the debuggee,
a debuggee opcode is temporarily replaced by an instruction causing BREAKPOINT exception (the "INT 3" instruction on Intel ix86). When "INT 3" is executed, the BREAKPOINT handler is executed. Otherwise, the exception processing is the same than the one described in previous section.

8 Conclusion

In this document we have presented how we envisage to add remote debugging facilities to RTEMS by implementing a remote debugger daemon for GDB. As any debug implemented in software, it will have limitation but we are confident that most of them can be removed by adding separate software components dedicated to debug activity. We must keep in mind that even with this approach, no software will enable the debug of code with interrupt entirely masked at processor level and that In Circuit Emulator (ICE) or use of BDM extension on the target board are the ultimate way to really debug any portion of an RTOS. BDM support in GDB is still weak but people are working on it and we may get something better in a near future.

Figure 5: remote debugger tasks and handlers

19