GNAT User's Guide

Supplement for Cross Platforms

GNAT, The GNU Ada Compiler
GNAT GPL Edition, Version 2012
Document revision level 247113
Date: 2012/03/28

AdaCore

About This Guide

This guide describes the use of GNAT, a compiler and software development toolset for the full Ada programming language, in a cross compilation environment. It supplements the information presented in the *GNAT User's Guide*. It describes the features of the compiler and tools, and details how to use them to build Ada applications that run on a target processor.

GNAT implements Ada 95 and Ada 2005, and it may also be invoked in Ada 83 compatibility mode. By default, GNAT assumes Ada 2005, but you can override with a compiler switch to explicitly specify the language version. (Please refer to the section "Compiling Different Versions of Ada", in *GNAT User's Guide*, for details on these switches.) Throughout this manual, references to "Ada" without a year suffix apply to both the Ada 95 and Ada 2005 versions of the language.

This guide contains some basic information about using GNAT in any cross environment, but the main body of the document is a set of Appendices on topics specific to the various target platforms.

What This Guide Contains

This guide contains the following chapters:

- About This Guide
- Chapter 1 [Preliminary Note for Cross Platform Users], page 5, describes the basic differences between the cross and native versions of GNAT.
- Appendix A [Bareboard and Custom Kernel Topics], page 7, describes how to build, load and run programs on a target without requiring the services of an underlying kernel.
- Appendix B [Common VxWorks Topics], page 15, presents information relevant to all VxWorks targets for cross-compilation.
- Appendix C [Tornado 2.x / VxWorks 5.x Topics], page 43, presents information relevant to Tornado 2.x and VxWorks 5.x targets.
- Appendix D [Workbench / VxWorks 6.x Topics], page 59, presents information relevant to Workbench and VxWorks 6.x targets.
- Appendix E [VxWorks 653 Topics], page 67, presents information relevant to VxWorks 653 targets.
- Appendix F [VxWorks MILS Topics], page 87, presents information relevant to VxWorks MILS targets.
- Appendix G [LynxOS Topics], page 105, presents information relevant to the LynxOS target for cross-compilation configurations.
- Appendix H [AAMP Topics], page 111, presents information relevant to the AAMP target for cross-compilation configurations.

- Appendix I [PowerPC 55xx ELF Topics], page 115, presents information relevant to the PowerPC 55xx ELF targets for cross-compilation configurations.
- Appendix J [AVR Topics], page 119, presents information relevant to the AVR microcontroller targets for cross-compilation configurations.
- Appendix K [LEON / ERC32 Topics], page 123, presents information relevant to the LEON / ERC32 targets for cross-compilation configurations.
- Appendix L [ELinOS Topics], page 135, presents information relevant to the ELinOS targets for cross-compilation configurations.
- Appendix M [PikeOS Topics], page 137, presents information relevant to the PikeOS targets for cross-compilation configurations.
- Appendix N [Customized Ravenscar Library Topics], page 151, presents information relevant to the customization of the Ravenscar run-time library on PowerPC.

What You Should Know before Reading This Guide

This user's guide assumes a basic familiarity with the Ada 95 language, as described in the International Standard ISO/IEC-8652:1995, January 1995. It does not require knowledge of the new features introduced by Ada 2005, (officially known as ISO/IEC 8652:1995 with Technical Corrigendum 1 and Amendment 1). Both Ada reference manuals are included in the GNAT documentation package.

This user's guide also assumes that you are familiar with the *GNAT User's Guide*, and that you know how to use GNAT on a native platform.

Related Information

For further information, refer to the following documents:

- *GNAT User's Guide*, either the generic or VMS version; these describe the GNAT compilation model, software development environment, and toolset.
- *GNAT Reference Manual*, which contains all reference material for the GNAT implementation of Ada.
- *ASIS-for-GNAT User's Guide*, which describes how to use ASIS with the GNAT environment.
- *ASIS-for-GNAT Reference Manual*, which contains reference material supplementing the *ASIS-for-GNAT User's Guide*
- *Ada 95 Reference Manual*, which defines the Ada 95 language standard.
- *Ada 2005 Reference Manual*, which defines the Ada 2005 language standard.

- *Debugging with GDB*, which contains all details on the use of the GNU source-level debugger.
- *GNU Emacs Manual*, which contains full information on the extensible editor and programming environment Emacs.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `Functions`, `utility program names`, `standard names`, and `classes`.
- ‘`Option flags`’
- ‘`File Names`’, ‘`button names`’, and ‘`field names`’.
- *`Variables`*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

```
     and then shown this way.
```

Commands that are entered by the user are preceded in this manual by the characters “`$ `” (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the `$` replaced by whatever prompt character you are using.

1 Preliminary Note for Cross Platform Users

The use of GNAT in a cross environment is very similar to its use in a native environment. Most of the tools described in the *GNAT User's Guide*, including the ASIS-based tools (`gnatelim`, `gnatstub`, `gnatpp`), have similar functions and options in both modes. The major difference is that the name of the cross tools includes the target for which the cross compiler is configured. For instance, the cross `gnatmake` tool is invoked as *target*`-gnatmake` where *target* stands for the name of the cross target. Thus, in an environment configured for the target `powerpc-wrs-vxworks`, the `gnatmake` command is `powerpc-wrs-vxworks-gnatmake`. This convention allows the installation of a native and one or several cross development environments on the same machine.

An exception to the naming rule exists for the AAMP target (see Appendix H [AAMP Topics], page 111 for information on AAMP tool name conventions).

In the rest of this manual, we generally refer to the tools with their simple native names (e.g., `gcc`) when we are describing their generic functionality, and with their full cross names (e.g., *target*`-gcc`) when we are describing their invocation.

The tools that are most relevant in a cross environment are `gcc`, `gnatmake`, `gnatbind`, and `gnatlink` to build cross applications; and `gnatls` for cross library browsing.

`gdb` is usually available for cross debugging in text mode. The graphical debugger interface in `GPS` is always a native tool but it can be configured to drive a cross debugger, thus allowing graphical cross debugging sessions. Some other tools such as `gnatchop`, `gnatkr`, `gnatprep`, `gnatxref`, `gnatfind` and `gnatname` are also provided for completeness even though they do not differ greatly from their native counterparts.

The GNAT Project facility is integrated into the cross environment. GPS uses the `gnatlist` attribute in the `Ide` package, whose value is *target*`-gnatls`, to compute the cross-prefix. From this information the correct location for the GNAT run-time library, and thus also the correct cross-references, can be determined. See the GPS documentation for more details, in particular the section *Working in a Cross Environment*.

Appendix A Bareboard and Custom Kernel Topics

This chapter describes how to use the GNAT cross tools to build programs that do not require support from any kernel running on the target hardware. It uses the PowerPC with ELF object module format as the target platform to illustrate these issues.

A.1 Introduction

Throughout this chapter we will use one approach to linking. There are many ways that `gnatlink` (or `powerpc-elf-gnatlink` as it would be invoked on this platform), or `gnatmake` can be used; but we will adopt a lower level approach that is a bit clearer and more flexible.

The program `gnatlink` normally performs 2 steps:

1. it calls `gcc` to compile the file generated by `gnatbind`;
2. it calls `gcc` to link all the relocatable object files into an executable.

We will break out those steps. We will use `powerpc-elf-gcc` to compile the binder-generated file, and `powerpc-elf-ld` to do the linking.

Other configurations of GNAT, native or targeting an embedded OS, use tools for manipulating binary files that are provided with the target system (whether native or cross). In the platform illustrated in this chapter, because there is no target OS and no development tools that would be provided with an OS, GNAT provides the tools that are necessary or useful. GNAT Cross comes with with the GNU `binutils` package properly configured and built. The linker, `powerpc-elf-ld`, is one of those tools. .

`gcc` (or `powerpc-elf-gcc`) is really a driver program that runs other programs for compiling, assembling, linking and performing other needed steps. When used for linking, `gcc` knows which object files and libraries are needed for the target environment, and it adds these to the linker command line.

A.2 Examples

Given the many variations on the host and target environment that you may be using, it is not possible to formulate a universally applicable sequence of commands for building, loading, and running a program. Instead, this section presents a series of scenarios reflecting various levels of sophistication in the target tools, with enough detail to allow you to prepare the necessary setup. For illustration purposes the examples will assume a PowerPC / ELF target, and occasionally may reflect a Unix host.

There are two main issues that you need to consider:

- how an executable program is loaded onto the target hardware, and
- the state of the hardware / software environment after the program has been loaded.

One of the simpler configurations is as follows:

- the target hardware has a monitor program in ROM that runs at powerup
- a command line interface controls the target through a serial port connected to the host
- a load command in the monitor program uses `tftp` to load a specified ELF file from the host (there is an ethernet interface, and the monitor handles TCP/IP protocols)

In other words, the monitor on the target board is relatively "smart". To run a program that you've built with GNAT Cross (an executable ELF file) you just need to load the program and "go" – the monitor program keeps track of the start address recorded in the ELF file.

The following simple example will illustrate the issues with building, loading and running a program; it merely sets a word in RAM to a particular value. With the ROM monitor program you can see the program's effect by looking at the memory location before and after execution.

```
with System.Storage_Elements; use System.Storage_Elements;

procedure nu is
   J : Integer;
   for J'Address use To_Address (16#200_000#);
begin
   J := 5;
end nu;
```

Based on your experience with the GNAT tools for a native environment, you might try to build this program with the command:

```
powerpc-elf-gnatmake nu
```

While there is a small chance that you could get the resulting executable to run, this is unlikely. In the relatively easy-to-use setup that we've described so far, the load command will likely read the entry point from the ELF file '`nu`'. Since there is no symbol named `_start`, the linker has chosen the first address in the .text section as the entry point. However, that is not where the program should start execution. Moreover, the linker has chosen some builtin default addresses for placing the various sections of the program. It is highly unlikely that they will be appropriate for your target hardware.

Assuming that `16#100_000#` is a good starting location in RAM for your program, the following steps will build a more useful executable.

```
powerpc-elf-gnatmake -g -b -c nu
powerpc-elf-gcc -c b~nu.adb
```

```
powerpc-elf-ld -e main -Ttext 0x100000 b~nu.o nu.o -o nu
```

Note that the '`-b -c`' options to `gnatmake` combine compiling and binding, and omit the linking step.

As with the native tools, the binder creates a file named '`b~`*XXX*`.adb`' where *XXX* is the name of the main subprogram. The second command above compiles the binder output file. The `ld` command works as follows:

- the '`-e main`' option tells `ld` that the symbol `main` is the entry point
- the '`-Ttext 0x100000`' option tells `ld` to start the `.text` section at the address `16#100_000#`. We are using the knowledge that `.text` is the first segment in memory, and `ld` places other segments following it.

In this case, it is also possible to combine these steps into a single command:

```
powerpc-elf-gnatmake -g nu -largs -Wl,-e,main,-Ttext,0x100000
```

When the resulting `nu` is loaded, the image will start at `16#100_000#`, and the entry point will be wherever the procedure `main` is placed.

Although this is more likely to run than the earlier example, there still may be some problems. To get more detail you can use the '`-save-temps`' option for `gcc` (or '`-cargs -save-temps`' to `gnatmake`), which will save the intermediate assembly language file using the '`.s`' file extension. The symbol name for the outer level procedure has `_ada_` prepended and is thus `_ada_nu`. In '`b~nu.adb`' you will find the main procedure (notice the export in '`b~nu.ads`'). It sets default values for command line variables (which do not apply in this environment), calls `adainit` (which would run the elaboration code if there was any), calls `Break_Start` (which is there to provide a convenient place to set a breakpoint when debugging), and, finally, calls your `nu` code.

If your target environment is not so fortunately set up, you will not be able to use the ELF file `nu` produced above. Such a simple scheme also breaks down in numerous ways as soon as you write any code with any more complexity.

To cope with such issues, you will need to create and use a startup routine in assembly language. Define the symbol `_start` at the desired code entry point; the linker will find that symbol, and you no longer need the '`-e`' option to `ld`. Several additional items will also be useful to include in the startup code: zeroing `.bss`, and setting the stack pointer and perhaps other registers.

Uninitialized global data are reserved in the `.bss` section in the executable file. During linking, all uninitialized variables are assigned an address in `.bss`, and all references to these variables are relocated properly, but no actual data are contained in the ELF executable (it is allocatable but not loadable). It is expected that the locations within `.bss` are set to zero before execution begins.

We will no longer assume that some smart loader is taking care of so much; the startup code is responsible for zeroing the `.bss` data, before it branches to your application's Ada entry. Since `nu` contained no declarations of uninitialized variables (the address clause causes `J` to be treated differently), this was not an issue for that simple program. Also, it is conceivable that the ROM monitor will set the stack to a usable value before branching to the entry point. However, it is more likely that you will want to set the stack pointer early in your startup code, and there may be other registers that must be set. For example, if the relevant Application Binary Interface (ABI) specifies an addressing mode that is relative to a designated register, the linker will output a symbol containing the value that the register must be set to. For example, on MIPS, `_gp` must be loaded into register 28. On PowerPC you could set `r13` to point to the small data area (although this is not necessary unless you take advantage of this feature, and is not relevant here).

Here is the "bare bones" startup assembly code, '`start.s`':

```
        top_of_stack = 0x200000

        .section ".text"
        .global  _start
_start:
        lis 1, __bss_start@ha
        la  1, __bss_start@l (1)
        lis 2, __end@ha
        la  2, __end@l (2)
        li  3, 0
bssloop:
        cmp  0,0,1,2
        bge  bssdone
        stw  3, 0(1)
        addi 1, 1, 4
        b    bssloop
bssdone:
        addis 1, 0, top_of_stack@h
        ori   1, 1, top_of_stack@l

        bl    main
forever:
        b     forever
```

And here is the link command:

```
powerpc-elf-ld -Ttext 0x100000 -o nu start.o nu.o b~nu.o
```

A single command that would build the application is:

```
powerpc-elf-gnatmake -g nu -largs -Wl,-Ttext,0x100000,start.o
```

It is very likely that your setup may work with S-record files. If so, the following `objcopy` command converts the executable ELF file '`nu`' to an S-record file '`nu.sre`' (details are in the binutils documentation):

```
powerpc-elf-objcopy -S -O srec nu nu.sre
```

A script written in `ld`'s script language controls how `ld` composes the pieces of a relocatable ('`.o`') file when creating an executable. Up to now, the default builtin script has been assumed. You can see this default script by running `powerpc-elf-ld -verbose`, which writes the contents to your terminal. Note that this script is large and complicated, containing data specific to different debugging formats, programming languages (C++ in particular), and dynamic libraries (not applicable). However, you will also find the definitions of `__bss_start` and `__end`, which reflect the. bounds of the `.bss` section of the executable. Complete information about the linker and its scripting language may be found in the *Using ld* document.

An extremely simple linker script might be a better starting point for building what you need. Here is one that will work in our still relatively amenable working environment:

```
OUTPUT_FORMAT("elf32-powerpc", "elf32-powerpc",
              "elf32-powerpc")
OUTPUT_ARCH(powerpc)
ENTRY(_start)
PROVIDE (top_of_stack = 0x200000);
SECTIONS
{
  . = 0x100000;
  .text :
  {
    *(.text)
    *(.rodata)
  }

  .data :
  {
    *(.data .sdata .sdata2)
  }

  __bss_start = .;
  .bss :
  {
   *(.bss .sbss)
   *(COMMON)
  }
  __end = .;
}
```

Briefly, this script defines three output segments:

- .text, which contains all of the `.text` and `.rodata` sections from the input files;
- .data
- .bss

"." is referred to as the location counter. It is set to `0x100000` before the first segment, and all of the code follows after that (since "." is never explicitly set to a new value). The location counter is used to set the values of the `__bss_start` and `__end` symbols. The script defines the symbol `top_of_stack`, so you may remove it from the startup code if you prefer to use the linker script to set its value. `PROVIDE` defines the value if it isn't defined elsewhere.

Unless you explicitly discard input sections, they will be copied to the output executable with the same name. E.g., if you compiled your code with debug information, linking with the script above will copy it into the executable – you will then be able to debug.

A note on debugging: the version of `gdb` provided with GNAT Cross is "Ada aware". But you need to determine how `gdb` running on your host will interact with code on your target. Most likely, it will use `gdb`'s serial protocol to interact with some agent that can control execution of your code. One example is the use of a hardware probe that controls the processor through a JTAG (BDM, COPS) port. GNAT Cross supports the Abatron BDI 2000, which is a hardware device that connects to the target through JTAG, implements the `gdb` serial protocol in firmware, and communicates using that protocol over its ethernet port. For example, a BDI could be known on your local network by the hostname `bdi`, and when running `powerpc-elf-gdb` you would use the `gdb` command `target remote bdi:2001`. This tells `gdb` that it will control the execution of your program by using the serial protocol running on port `2001` of the host `bdi`. Another possibility is that there is a kernel running on your target hardware that controls program execution (your application) and which implements the `gdb` serial protocol over some connection – often a serial port.

We will highlight one other feature of `ld` scripting which is often used, particularly in later stages of development. Early on, you might have a way of loading your program into RAM and running it from there. Eventually, you might want to burn some parts into ROM and (particularly) if your code runs from hardware bootup you will have an issue with the `.data` segment. At bootup, the initial data in `.data` is in ROM at one range of addresses, but it has to be in RAM during execution, at a different range of addresses. (You might well want `.text` moved to RAM also for performance reasons, but we'll illustrate with `.data`.) `ld` assumes that every section has two addresses: the VMA, or "virtual memory address", and the LMA, or "load module address". The VMA is the run-time address of the beginning of the section in RAM; the LMA is the address in ROM. In the `ld` script example, there was nothing specified between the name of the output section `.data` and the following : character. The syntax allows you to specify the starting address (VMA) there. It defaults to the location counter if not specified. By default, LMA is set to the VMA. An `AT` clause is used to set the LMA separately. For example:

```
data_image_begin = 0xdc000000;
data_begin       = 0xE0000000;
```

```
  .data data_begin : AT (data_image_begin) {
    *(.data .sdata .sdata2)
  }
  data_end = .;
```

In this description of `.data`, the section is loaded at `data_image_begin`, but is expected to be at `data_begin` at run time. I.e., all references to objects in `.data` are resolved to addresses in the section starting at the VMA, `data_begin`. Code can be added to '`start.s`' to use the three symbols defined to copy the `.data` segment from ROM to RAM before the application starts up.

Some examples are supplied in the '`examples`' directory of the GNAT installation containing multi-language applications with project files. A '`Makefile`' is provided to build the source code for the desired target and run time. For example:

```
make TARGET=powerpc-elf RUNTIME=zfp
```

A.3 Development Support

Sometimes, starting a development on a bare board platform is difficult because of the lack of functionality for debugging and early verification. For example, text output routines are not expected to be used in the final version of the software, and hence they are not available in certifiable run-time libraries, but they are useful to display execution logs.

The cross-compilation toolchains come with a support library (`zfp_support`) including text output routines, image attributes, memory operations, and some others, intended to be used during the development and debugging phases.

To use this library, you just need to add `with "zfp_support";` to the top of your project file. If you want to write a simple `Hello world` example for PowerPC that can be run on QEMU, you can write the program simply as:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Hello is
begin
   Put_Line ("Hello");
end Hello;
```

A very simple project file would be the following:

```
with "zfp_support.gpr";

project hello is
   for Main use ("hello.adb");
end hello;
```

And then, building and executing the example can be achieved with the following sequence of commands:

```
$ powerpc-elf-gnatmake -P hello -XVARIANT=powerpc-elf-qemu
$ powerpc-elf-qemu hello
Hello
```

Appendix B Common VxWorks Topics

This chapter describes topics that are relevant to all GNAT for VxWorks configurations. Note that the GNAT tool prefix for Tornado 2.x / VxWorks 5.x or Workbench / VxWorks 6.x is `powerpc-wrs-vxworks-`, `e500v2-wrs-vxworks-` or `i586-wrs-vxworks-`; for Workbench / VxWorks 653, it is `powerpc-wrs-vxworksae-`; for VxWorks MILS, it is `powerpc-wrs-vxworksmils-` or `e500v2-wrs-vxworksmils-`.

B.1 Executing a Program on VxWorks

Getting a program to execute involves loading it onto the target, running it, and then (if re-execution is needed) unloading it. These instructions apply mainly to Tornado 2.x / VxWorks 5.x, but can also apply to Workbench / VxWorks 653 when building an application for the Module OS, or Workbench / VxWorks 6.x when building an application for the kernel space. (Note that this is not the usual way of executing applications on VxWorks 653 - see the sections on VxWorks 653 for details).

B.1.1 Loading and Running the Program

An Ada program is loaded and run in the same way as a C program. Details may be found in the *Tornado User's Guide*.

In order to load and run our sample program, we assume that the target has access to the disk of the host containing the required object and that its working directory has been set to the directory containing this object. The commands are typed in Tornado's Windshell. The `windsh` prompt is the `->` sequence. In this example, the object is named `hello`, and when its execution results in the display of the `Hello World` string.

For VxWorks 5.x and VxWorks 6.x:

```
-> vf0=open("/vio/0",2,0)
new symbol "vf0" added to symbol table.
vf0 = 0x2cab48: value = 12 = 0xc
-> ioGlobalStdSet(1,vf0)
value = 1 = 0x1
-> ld < hello
value = 665408 = 0xa2740
-> hello
Hello World
value = 0 = 0x0
->
```

For VxWorks 653, in the Module OS, the load command is different:

```
[vxKernel] -> vf0=open("/vio/0",2,0)
new symbol "vf0" added to symbol table.
vf0 = 0x2cab48: value = 12 = 0xc
```

```
[vxKernel] -> ioGlobalStdSet(1,vf0)
value = 1 = 0x1
[vxKernel] -> ml < hello
value = 665408 = 0xa2740
[vxKernel] -> hello
Hello World
value = 0 = 0x0
[vxKernel] ->
```

The first two commands redirect output to the shell window. They are only needed if the target server was started without the `-C` option. The third command loads the module, which is the file '`hello`' created previously by the `gnatmake` command. Note that for VxWorks 653, the `ml` command replaces `ld`.

The `Hello World` program comprises a procedure named `hello`, and this is the name entered for the procedure in the target server's symbol table when the module is loaded. To execute the procedure, type the symbol name `hello` into `windsh` as shown in the last command above.

Note that by default the entry point of an Ada program is the name of the main Ada subprogram in a VxWorks environment. It is possible to use an alternative name; see the description of `gnatbind` options for details.

B.1.2 Unloading the Program

It is important to remember that you must unload a program once you have run it. You cannot load it once and run it several times. If you don't follow this rule, your program's behavior can be unpredictable, and will most probably crash.

This effect is due to the implementation of Ada's *elaboration* semantics. The unit elaboration phase comprises a *static* elaboration and a *dynamic* elaboration. On a native platform they both take place when the program is run. Thus rerunning the program will repeat the complete elaboration phase, and the program will run correctly.

On VxWorks, the process is a bit different. The static elaboration phase is handled by the loader (typically when you type `ld < program_name` in `windsh`). The dynamic phase takes place when the program is run. If the program is run twice and has not been unloaded and then reloaded, the second time it is run, the static elaboration phase is skipped. Variables initialized during the static elaboration phase may have been modified during the first execution of the program. Thus the second execution isn't performed on a completely initialized environment.

Note that in C programs, elaboration isn't systematic. Multiple runs without reload might work, but, even with C programs, if there is an elaboration phase, you will have to unload your program before re-running it.

B.2 Mixed-Language Programming

The *GNAT User's Guide*, in the section *Building Mixed Ada & C++ Programs*, subsection "A Simple Example", presents the compilation commands needed to build one of the GNAT examples. The relevant commands for the GNAT VxWorks/PowerPC target are as follows, assuming WIND_BASE is correctly defined:

```
$ powerpc-wrs-vxworks-gnatmake -c simple_cpp_interface
$ powerpc-wrs-vxworks-gnatbind -n simple_cpp_interface
$ powerpc-wrs-vxworks-gnatlink simple_cpp_interface -o ada_part
$ c++ppc -c -DCPU=PPC604  -I$WIND_BASE/target/h -I$WIND_BASE/target/h/wrn/coreip cpp_main.C
$ c++ppc -c -DCPU=PPC604  -I$WIND_BASE/target/h -I$WIND_BASE/target/h/wrn/coreip ex7.C
$ ldppc -r -o my_main my_main.o ex7.o ada_part
```

Note that use of gprconfig/gprbuild will insert the relevant includes above automatically.

The GNU C and C++ compilers for VxWorks 653 and for the VxWorks MILS VxWorks and High Assurance Environment (HAE) Guest OS's are compatible with the full, cert (653 only) and ravenscar-cert run-time libraries with respect to exception propagation. All of these compilers use the "setjmp/longjmp" (SJLJ) exception propagation mechanism. Note that for VxWorks MILS, the VxWorks Guest OS C and C++ compilers match those used for VxWorks 653; the HAE C and C++ compilers match those provided with VxWorks 6.6.

VxWorks 5 GNU C has the same compatibility with the default (SJLJ) library.

VxWorks 6.6 and VxWorks Cert 6.6 and lower GNU C and C++ compilers are also compatible with SJLJ. Note that as of GNAT 6.3.1, SJLJ is no longer the default run-time library, so must be selected explicitly with builder switch `--RTS=kernel-sjlj`.

VxWorks 6.7 and higher GNU C and C++ compilers are compatible with the "zero cost exception handling" scheme that is now the default GNAT run-time library. As of this writing, there are incompatibilities in obtaining tracebacks across language boundaries.

"Munching", the process of calling C++ static initializers, is automatic when building applications under Workbench 3.2 and higher. The Workbench application project types build the necessary wrapper around the application generated by GNAT. For VxWorks 5 applications that include C++ sources, munching must be done manually. See the Wind River documentation.

B.3 Kernel Configuration for VxWorks

When configuring your VxWorks kernel we recommend including the target shell. If you omit it from the configuration, you may get undefined symbols at load time, e.g.

```
-> ld < hello.exe
```

```
Loading hello.exe
Undefined symbols:
mkdir
```

Generally, such undefined symbols are harmless since these are used by optional parts of the GNAT run time. However if running your application generates a VxWorks exception or illegal instruction, you should reconfigure your kernel to resolve these symbols.

B.4 Kernel Compilation Issues for VxWorks

This section describes the different ways to statically link an Ada module (built using the GNAT toolchain) with a VxWorks kernel. The main issue is that two distinct toolchains are used:

- The Wind River toolchain (diab or gnu)
- The GNAT toolchain

Both toolchains compile Ada or C modules using either of two libraries: GCC Library or Wind River Library. These two libraries share common symbols and are mostly compatible. However, there is a potential issue with the exception handling functions. If you tried to directly link your Ada module with the VxWorks kernel you would get "duplicate symbols" errors for the linker, as two versions of the exceptions management library are included. In order to resolve this issue you will need, depending on the context, to select only one of these libraries or else find a way to include both of them. This section covers two topics:

- Which library to use
- How to achieve the link

The library to select depends on the following issues:

- Whether C++ is used
- Whether the ZCX run-time is used for the Ada module

If you are using C++ in your project, then you will need the C++ exception handling functions. As the GCC Library from the GNAT toolchain implements the exception mechanisms differently from the Wind River Library, you need in this case to use the Wind River Library.

The ZCX run-time library packaged in the PowerPC GNAT toolchains uses the exception mechanism implemented in the GCC Library distributed with GNAT. If you are using this run-time library you must select the GCC Library and not the one from the Wind River toolchain.

The sjlj Ada run-time library in the GNAT toolchain does not rely on the mechanism present in the GCC Library, so if you are using that run-time in your Ada module, it's better to use the Wind River Library as it allows you to use C++.

If you are using both C++ and the ZCX runtime, then the situation is more complex as you need the two libraries. A possible approach is described at the end of this section.

There are several different ways to perform the link:

- Using the Wind River Library

  You need to compile your Ada module with the '`-nostdlib`' option, which excludes the GNAT GCC Library:

  ```
  gnatmake foo.adb -largs -nostdlib
  ```

  and then add the resulting executable file ('`foo`' or '`foo.exe`') to your kernel. On VxWorks 5.x this can be achieved by adding the object in the `EXTRA_MODULES` macro of your kernel project.
- Using the GNAT GCC Library

  You need to do the following:
  - Compile your Ada module:

    ```
    gnatmake foo.adb -largs -nostdlib
    ```
  - Add the resulting executable file to the kernel as described above.
  - Remove the Wind River Library from the kernel and then put in the GNAT version instead. On VxWorks 5.x, this step is achieved by modifying the `LIBS` macro. In order to obtain the location of the GNAT GCC Library do the following:

    ```
    powerpc-wrs-vxworks-gcc -print-libgcc-file-name <options>
    ```

    Even if this command does not perform any compilation it is important to specify to GCC the options used when the Ada module was compiled. Indeed, the GNAT toolchain provides several versions of the GCC Library; by specifying the GCC options in the previous command you will obtain the correct version.
  - In your kernel configuration, be sure to remove "Compiler Support Routines" components. Indeed this component adds a C file to your kernel that will create a dependence on all '`.o`' files present in the Wind River Library, and you will receive error messages from the linker about missing symbols. An inconvenient consequence is that if you plan to dynamically download modules to your kernel, then some symbols from the GCC Library might not be present.
- Including both GCC and Wind River Libraries
  - Compile your Ada module

    ```
    powerpc-wrs-vxworks-gnatmake foo.adb
    ```
  - Make all the symbols of your Ada module local except the main entry point

    ```
    objcopyppc -G foo foo new-foo
    ```

- Add the resulting executable file to your kernel.

If you need a bootable application, then refer to the section *Creating Bootable Application* in the *Projects* chapter of the *Tornado User's Guide*. There is a detailed discussion on adding user initialization routines.

B.5 Main Task Attributes and the Application Stub

Several characteristics of the application main can only be controlled from the VxWorks environment: main stack size, main priority, and main task attributes. Often the default values are fine. In other cases one must explicitly set the desired values outside of the application source code.

The system call `taskSpawn()` can be used to provide explicit values for the task name string, its VxWorks priority, stack size and other options (such as the floating point type).

For applications requiring modification of these attributes that are invoked from the host or target shell, the system call can be used directly to spawn the application main task.

For applications started using a Workbench launcher, these attributes can be set in the launcher.

For applications being linked into a kernel image, one should use an application stub for this purpose. Although it can have any name, the default is `usrAppInit()` and is placed in a file in the project called '`usrAppInit.c`'. By defining the build macro USER_APPL_INIT one can use a routine with a different stub or go directly to the Ada main subroutine if no attribute modifications are needed.

Here is a sample usrAppInit.c suitable for starting an Ada application linked into a VxWorks kernel image:

```
#include <vxWorks.h>
#include <taskLib.h>

/* Ada binder-generated main - ADA_MAIN is passed as a macro from the makefile */

extern void ADA_MAIN(void);

void usrAppInit()

{

        int stackSize = 0x80000;

         /* For e500v2 processors, use VX_SPE_TASK instead */
        int spawnFlags = VX_FP_TASK;
```

```
          /* MAIN_TASK_NAME is passed as a macro from the makefile */
          char * mainTaskName = (char *) MAIN_TASK_NAME;

          /* VxWorks priority = 255 - Ada priority */
          int priority = 100;

          /* Spawn Ada environment task */

          taskSpawn(mainTaskName, priority, spawnFlags, stackSize,
                         (FUNCPTR) ADA_MAIN, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
}
```

ADA_MAIN is the name of the Ada main subprogram in lower case (i.e. the symbol for the entry point of the Ada application). MAIN_TASK_NAME is the name you would like associated with the Ada environment task for the application. These are typically passed using a -D<symbol>=<value> switch to the compiler in the makefile for the application component.

The stack size can be varied to suit the needs of the environment task. It must be spawned with at least option VX_FP_TASK or one of the other floating point type options, depending on your board.

B.6 Interrupt Handling for VxWorks

GNAT offers a range of options for hardware interrupt handling on VxWorks 5, for VxWorks 6 downloadable kernel modules (DKM) and for the VxWorks MILS VxWorks Guest OS.

Directly vectored interrupt procedure handlers can also be used in the CoreOS of VxWorks 653; otherwise interrupt handling is unsupported for VxWorks 653 and for VxWorks 6 real-time processes. These are operating system restrictions.

See *VxWorks MILS Topics* for configuration requirements for handling interrupts in a virtual board.

In rough order of latency and lack of restrictions:

- Directly vectored interrupt procedure handlers
- Directly vectored interrupt procedures that signal a task using a suspension object
- Ada 95 / Ada 2005 protected procedure handlers for interrupts
- Ada 83 style interrupt entry handlers for interrupts

In general, the range of possible solutions trades off latency versus restrictions in the handler code. Restrictions in direct vectored interrupt handlers are documented in the *VxWorks Programmer's Guide*. Protected procedure handlers have only the restriction that no potentially blocking operations are performed within the handler. Interrupt entries have no restrictions. We recommend the use of the protected procedure mechanism as providing the best balance of these considerations for most applications.

All handler types must explicitly perform any required hardware cleanups, such as issuing an end-of-interrupt if necessary.

For VxWorks 653, applications that handle interrupts must be loaded into the kernel protection domain (CoreOS).

Note that in the examples below certain routines are specific to VME bus interrupt handling. These are only examples, and the appropriate routines for your processor and board should be used instead.

- Direct Vectored Interrupt Routines

  This approach provides the lowest interrupt latency, but has the most restrictions on what VxWorks and Ada run-time calls can be made, as well as on what Ada entities are accessible to the handler code. Such handlers are most useful when there are stringent latency requirements, and very little processing is to be performed in the handler. Access to the necessary VxWorks routines for setting up such handlers is provided in the package `Interfaces.VxWorks`.

  VxWorks restrictions are described in the *VxWorks Programmer's Manual*. Note in particular that floating point context is not automatically saved and restored when interrupts are vectored to the handler. If the handler is to execute floating point instructions, the statements involved must be bracketed by a pair of calls to `fppSave` and `fppRestore` defined in `Interfaces.VxWorks`. As the compiler may use floating point registers for non floating point operation (such as memory block copy) it is highly recommended to either save and restore floating point registers or to inspect code generated for the interrupt handler.

  As stack checking doesn't work within an interrupt handler, be sure that units containing interrupt handlers are not compiled with '`-fstack-check`'.

  As there is no task identity when executing an interrupt handler, any Ada run time library code that depends on knowing the current task's identity must not be used. This includes tasking constructs (except for most subprograms of `Ada.Synchronous_Task_Control`), concatenation and other functions with unconstrained results and exceptions propagation.

  In general, it is a good idea to save and restore the handler that was installed prior to application startup. The routines `intVecGet` and `intVecSet` are used for this purpose. The Ada handler code is installed into the vector table using routine `intConnect`, which generates wrapper code to save and restore registers.

  Example:

```
with Interfaces.VxWorks; use Interfaces.VxWorks;
with System;

package P is

   Count : Natural := 0;
   pragma Atomic (Count);

   -- Interrupt level used by this example
   Level : constant := 1;

   -- Be sure to use a reasonable interrupt number for the target
   -- board! Refer to the BSP for details.
   Interrupt : constant := 16#14#;

   procedure Handler (Parameter : System.Address);

end P;

package body P is

   procedure Handler (parameter : System.Address) is
      S : Status;
   begin
      Count := Count + 1;
      -- Acknowledge interrupt. Not necessary for all interrupts.
      S := sysBusIntAck (intLevel => Level);
   end Handler;
end P;

with Interfaces.VxWorks; use Interfaces.VxWorks;
with Ada.Text_IO; use Ada.Text_IO;

with P; use P;
procedure Useint is
   task T;

   S : Status;

   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("Generating an interrupt...");
         delay 1.0;

         -- Generate interrupt, using interrupt number
         S := sysBusIntGen (Level, Interrupt);
      end loop;
   end T;
```

```
      --  Save old handler
      Old_Handler : VOIDFUNCPTR := intVecGet (INUM_TO_IVEC (Interrupt));
   begin
      S := intConnect (INUM_TO_IVEC (Interrupt), Handler'Access);
      S := sysIntEnable (intLevel => Level);

      for I in 1 .. 10 loop
         delay 2.0;
         Put_Line ("value of count:" & P.Count'Img);
      end loop;

      --  Restore previous handler
      S := sysIntDisable (intLevel => Level);
      intVecSet (INUM_TO_IVEC (Interrupt), Old_Handler);
   end Useint;
```

- Direct Vectored Interrupt Routines

  A variation on the direct vectored routine that allows for less restrictive handler code is to separate the interrupt processing into two levels.

  The first level is the same as in the previous section. Here we perform simple hardware actions and signal a task pending on a Suspension_Object (defined in `Ada.Synchronous_Task_Control`) to perform the more complex and time-consuming operations. The routine `Set_True` signals a task whose body loops and pends on the suspension object using `Suspend_Until_True`. The suspension object is declared in a scope global to both the handler and the task. This approach can be thought of as a slightly higher-level application of the `C` example using a binary semaphore given in the VxWorks Programmer's Manual. In fact, the implementation of `Ada.Synchronous_Task_Control` is a very thin wrapper around a VxWorks binary semaphore.

  This approach has a latency between the direct vectored approach and the protected procedure approach. There are no restrictions in the Ada task code, while the handler code has the same restrictions as any other direct interrupt handler.

  Example:

```
   with System;
   package Sem_Handler is

      Count : Natural := 0;
      pragma Atomic (Count);

      --  Interrupt level used by this example
      Level : constant := 1;
      Interrupt : constant := 16#14#;

      --  Interrupt handler providing "immediate" handling
      procedure Handler (Param : System.Address);

      --  Task whose body provides "deferred" handling
      task Receiver is
          pragma Interrupt_Priority
             (System.Interrupt_Priority'First + Level + 1);
       end Receiver;

   end Sem_Handler;

   with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
   with Interfaces.VxWorks; use Interfaces.VxWorks;
   package body Sem_Handler is

      SO : Suspension_Object;

      task body Receiver is
      begin
         loop
            --  Wait for notification from immediate handler
            Suspend_Until_True (SO);

            --  Interrupt processing
            Count := Count + 1;
         end loop;
      end Receiver;

      procedure Handler (Param : System.Address) is
         S : STATUS;
      begin
         --  Hardware cleanup, if necessary
         S := sysBusIntAck (Level);

         --  Signal the task
         Set_True (SO);
      end Handler;

   end Sem_Handler;

   with Interfaces.VxWorks; use Interfaces.VxWorks;
   with Ada.Text_IO; use Ada.Text_IO;
   with Sem_Handler; use Sem_Handler;
   procedure Useint is

      S : STATUS;

      task T;

      task body T is
```

```
      begin
         for I in 1 .. 10 loop
            Put_Line ("Generating an interrupt...");
            delay 1.0;

            --  Generate interrupt, using interrupt number
            S := sysBusIntGen (Level, Interrupt);
         end loop;
      end T;

      --  Save old handler
      Old_Handler : VOIDFUNCPTR := intVecGet (INUM_TO_IVEC (Interrupt));
   begin
      S := intConnect (INUM_TO_IVEC (Interrupt), Handler'Access);
      S := sysIntEnable (intLevel => Level);

      for I in 1 .. 10 loop
         delay 2.0;
         Put_Line ("value of Count:" & Sem_Handler.Count'Img);
      end loop;

      --  Restore handler
      S := sysIntDisable (intLevel => Level);
      intVecSet (INUM_TO_IVEC (Interrupt), Old_Handler);
      abort Receiver;
   end Useint;
```

- Protected Procedure Handlers for Interrupts

  This is the recommended default mechanism for interrupt handling. It essentially wraps the hybrid handler / task mechanism in a higher-level abstraction, and provides a good balance between latency and capability.

  Vectored interrupts are designated by their interrupt number, starting from 0 and ranging to the number of entries in the interrupt vector table - 1.

  In the GNAT VxWorks implementation, the following priority mappings are used:

  - Normal task priorities are in the range 0 .. 245.
  - Interrupt priority 246 is used by the GNAT `Interrupt_Manager` task.
  - Interrupt priority 247 is used for vectored interrupts that do not correspond to those generated via an interrupt controller.
  - Interrupt priorities 248 .. 255 correspond to PIC interrupt levels 0 .. 7.
  - Priority 256 is reserved to the VxWorks kernel.

  Except for reserved priorities, the above are recommendations for setting the ceiling priority of a protected object that handles interrupts, or the priority of a task with interrupt entries. It's a very good idea to follow these recommendations for vectored interrupts that come in through the

PIC as it will determine the priority of execution of the code in the protected procedure or interrupt entry.

No vectored interrupt numbers are reserved in this implementation, because dedicated interrupts are determined by the board support package. Obviously, careful consideration of the hardware is necessary when handling interrupts. The VxWorks BSP for the board is the definitive reference for interrupt assignments.

Example:

```
package PO_Handler is

   -- Interrupt level used by this example
   Level : constant := 1;

   Interrupt : constant := 16#14#;

   protected Protected_Handler is
      procedure Handler;
      pragma Attach_Handler (Handler, Interrupt);

      function Count return Natural;

      pragma Interrupt_Priority (248);
   private
      The_Count : Natural := 0;
   end Protected_Handler;

end PO_Handler;

with Interfaces.VxWorks; use Interfaces.VxWorks;
package body PO_Handler is

   protected body Protected_Handler is

      procedure Handler is
         S : Status;
      begin
         -- Hardware cleanup if necessary
         S := sysBusIntAck (Level);

         -- Interrupt processing
         The_Count := The_Count + 1;
      end Handler;

      function Count return Natural is
      begin
         return The_Count;
      end Count;
   end Protected_Handler;

end PO_Handler;

with Interfaces.VxWorks; use Interfaces.VxWorks;
with Ada.Text_IO; use Ada.Text_IO;

with PO_Handler; use PO_Handler;
procedure Useint is

   task T;

   S : STATUS;

   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("Generating an interrupt...");
         delay 1.0;

         --  Generate interrupt, using interrupt number
         S := sysBusIntGen (Level, Interrupt);
      end loop;
   end T;

begin
   S := sysIntEnable (intLevel => Level);

   for I in 1 .. 10 loop
      delay 2.0;
      Put_Line ("value of count:" & Protected_Handler.Count'Img);
   end loop;

   S := sysIntDisable (intLevel => Level);
end Useint;
```

This is obviously significantly higher-level and easier to write than the previous examples.

- Ada 83 Style Interrupt Entries

  GNAT provides a full implementation of the Ada 83 interrupt entry mechanism for vectored interrupts. However, due to latency issues, we only recommend using these for backward compatibility. The comments in the previous section regarding interrupt priorities and reserved interrupts apply here.

  In order to associate an interrupt with an entry, GNAT provides the standard Ada convenience routine `Ada.Interrupts.Reference`. It is used as follows:

```
Interrupt_Address : constant System.Address :=
   Ada.Interrupts.Reference (Int_Num);

task Handler_Task is
   pragma Interrupt_Priority (248);  -- For instance
   entry Handler;
   for Handler'Address use Interrupt_Address;
end Handler_Task;
```

Since there is no restriction within an interrupt entry on blocking operations, be sure to perform any hardware interrupt controller related operations before executing a call that could block within the entry's accept statements. It is assumed that interrupt entries are always open alternatives when they appear within a selective wait statement. The presence of a guard gives undefined behavior.

Example:

```
with Ada.Interrupts;
with System;
package Task_Handler is

   --  Interrupt level used by this example
   Level : constant := 1;

   Interrupt : constant := 16#14#;

   Interrupt_Address : constant System.Address :=
      Ada.Interrupts.Reference (Interrupt);

   task Handler_Task is
      pragma Interrupt_Priority (248);  -- For instance
      entry Handler;
      for Handler'Address use Interrupt_Address;

      entry Count (Value : out Natural);
   end Handler_Task;
end Task_Handler;

with Interfaces.VxWorks; use Interfaces.VxWorks;
package body Task_Handler is

   task body Handler_Task is
      The_Count : Natural := 0;
      S : STATUS;
   begin
      loop
         select
            accept Handler do
               --  Hardware cleanup if necessary
               S := sysBusIntAck (Level);

               --  Interrupt processing
               The_Count := The_Count + 1;
            end Handler;
         or
            accept Count (Value : out Natural) do
               Value := The_Count;
            end Count;
         end select;
      end loop;
   end Handler_Task;

end Task_Handler;

with Interfaces.VxWorks; use Interfaces.VxWorks;
with Ada.Text_IO; use Ada.Text_IO;

with Task_Handler; use Task_Handler;
procedure Useint is

   task T;

   S : STATUS;
   Current_Count : Natural := 0;

   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("Generating an interrupt...");
         delay 1.0;

         --  Generate interrupt, using interrupt number
         S := sysBusIntGen (Level, Interrupt);
      end loop;
   end T;

begin
   S := sysIntEnable (intLevel => Level);

   for I in 1 .. 10 loop
      delay 2.0;
      Handler_Task.Count (Current_Count);
      Put_Line ("value of count:" & Current_Count'Img);
   end loop;

   S := sysIntDisable (intLevel => Level);
   abort Handler_Task;
end Useint;
```

B.7 Handling Relocation Issues for PowerPc Targets

Under certain circumstances, loading a program onto a PowerPC board will fail with the message *Relocation value does not fit in 24 bits*. This section summarizes why and when such a problem will arise, and explains how it can be solved.

B.7.1 Background and Summary

Prior to release 3.15 of GNAT, the compiler's default behavior was to use relative addressing mode for all subprogram calls, including those in the GNAT run-time library. This led to the "24 bits" problem for many user applications. Starting with 3.15, the run-time library accompanying the compiler has been compiled with the '`-mlongcall`' command line argument, which directs the compiler to generate absolute addresses for subprogram calls. Furthermore, users who were encountering the problem started compiling their code with an explicit '`-mlongcall`' option, and the problem largely disappeared.

Starting with release 6.0.2 of GNAT, the effects of '`-mlongcall`' are turned on by default for all compilations, including the compilation of binder-generated files. Thus users no longer have to supply the '`-mlongcall`' option themselves.

In light of this change in the default behavior, a user requiring the relative addressing mode (for example for performance tuning) will now need to explicitly specify the '`-mno-longcall`' option.

B.7.2 Technical Details

VxWorks on the PowerPC follows the variation of the SVR4 ABI known as the Embedded ABI (*EABI*). In order to save space and time in embedded applications, the EABI specifies that the default for subprogram calls should be the branch instruction with relative addressing using an immediate operand. The immediate operand to this instruction (relative address) is 24 bits wide. It is sign extended and `2#00#` is appended for the last 2 bits (all instructions must be on a 4 byte boundary).

The resulting 26 bit offset means that the target of the branch must be within +/- 32 Mbytes of the PC-relative branch instruction. When loading a program, VxWorks completes the linking phase by resolving all of the unresolved references in the object being loaded. When one of those references is a relative address in a branch instruction, and the linker determines that the target is more than 32 Mbytes away from the branch, the error occurs.

This only happens when the BSP is configured to use more than 32 MBytes of memory. The VxWorks kernel is loaded into low memory addresses, and the error usually occurs when the target loader is used (because it loads objects into high memory, and thus calls from the program to the VxWorks kernel can be too far).

One way to solve this problem is to use the Tornado / Workbench host loader; this will place programs in low memory, close to the kernel.

For versions of GNAT which precede 6.0.2, the '`-mlongcall`' option for gcc causes the compiler to construct the absolute address of a subprogram in a register and use a branch instruction with absolute addressing mode. Starting with 6.0.2, absolute addresses are the default, and '`-mno-longcall`' is needed to use relative addresses for subprogram calls.

For GNAT versions 3.15 through 6.0.1, the GNAT run-time libraries are compiled with '`-mlongcall`'. In many cases the use of these libraries is sufficient to avoid the relocation problem, since it is the run-time library that contains calls to the VxWorks kernel that need to span the address space gap. The run-time libraries for version 6.0.2 and later also contain only the absolute addressing mode.

When using the GNAT 3.15 through 6.0.1, you may need to compile your application code using '`-mlongcall`' if there are calls directly to the kernel, if the application is very large, or in some specialized linking/loading scenarios.

You can compile individual files with `-mlongcall` by placing this option on the `gcc` command line (for brevity we are omitting the `powerpc-wrs-vxworks-` prefix on the commands shown in this paragraph). If you provide `-mlongcall` as an option for `gnatmake`, it will be passed to all invocations of `gcc` that `gnatmake` directly performs. Note that one other compilation is made by `gnatlink`, on the file created by `gnatbind` for the elaboration package body (see *Binding Using* `gnatbind` in the *GNAT User's Guide*). Passing `-mlongcall` to `gnatlink`, either directly on the `gnatlink` command line or by including `-mlongcall` in the `-largs` list of `gnatmake`, will direct `gnatlink` to compile the binder file with the `-mlongcall` option.

To see the effect of `-mlongcall`, consider the following small example:

```
procedure Proc is
   procedure Imported_Proc;
   pragma Import (Ada, Imported_Proc);
begin
   Imported_Proc;
end;
```

If you compile `Proc` with the default options (no `-mlongcall`) with a GNAT release prior to 6.0.2, the following code is generated:

```
_ada_proc:
        ...
        bl imported_proc
        ...
```

In contrast, here is the result with the defaults for GNAT 6.0.2 and later, or with the '`-mlongcall`' option passed to earlier releases

```
_ada_proc:
```

```
...
addis 9,0,imported_proc@ha
addi 0,9,imported_proc@l
mtlr 0
blrl
...
```

For additional information on this issue, please refer to WRS' SPRs 6040, 20257, and 22767.

B.8 Zero Cost Exceptions on PowerPC Targets

The Zero Cost eXceptions handling (ZCX) model is available for kernel mode on PowerPC processors.

On VxWorks 6, ZCX is activated by default. In this mode, proper transformation of signal deliveries into Ada exceptions requires increasing the kernel exception stack to at least 10*1024 bytes, which may be achieved by modifying the `taskKerExcStackSize` global variable before tasks are spawned. This affects all the tasks spawned after the change, and is typically needed for user stack-checking purposes. The setjmp/longjmp (SJLJ) exception model is still available, by switching to an alternative runtime library with '`--RTS=kernel-sjlj`'.

On VxWorks 5, the SJLJ model is activated by default. ZCX is available by switching to an alternative runtime library with '`--RTS=zcx`'.

B.9 Calling exported Ada procedures from the VxWorks shell

If you need to call Ada subprograms from outside your Ada application (for example, from the Windshell), you will in general need to make the executing task known to the Ada run-time library. A typical situation is the need to enable or disable traces in your Ada application from the shell by calling an Ada procedure that sets the value of a Boolean controlling the traces.

The problem is the following: when you call the Ada procedure, it may raise an exception on the target. This could be through normal execution, or it could be caused solely by the fact that the thread executing the procedure has not been made known to the Ada run-time library. There are a number of situations in full Ada applications where the run-time library must be able to determine what task is running, and must be able to access some Ada-specific data structures in order to do its job. If a VxWorks task has not been registered, the "id" that is accessed by the run-time library will be invalid, and will reference invalid data (or invalid memory).

The solution is to use the `GNAT.Threads.Register_Thread` and `GNAT.Threads.Unregister_Thread` routines, so that the task in which the called subprogram is executing (typically the shell task) is known by the Ada run-time. This way, when the Ada task id is requested at some point in the procedure's execution, a valid value will be fetched, and the necessary data structures will have been allocated and properly initialized.

The following small example shows how to use this feature:

```
package Traces is

   procedure Put_Trace (Info : String);
   --  Print Info on the standard output if the traces are on

   procedure Trace_On;
   pragma Export (C, Trace_On, "trace_on");
   --  Activate the traces

   procedure Trace_Off;
   pragma Export (C, Trace_Off, "trace_off");
   --  Deactivate the traces
end Traces;

with System;       use System;
with Ada.Text_IO;  use Ada.Text_IO;
with GNAT.Threads; use GNAT.Threads;
package body Traces is
   Trace : Boolean := False;

   ------------—-
   -- Put_Trace –
   ------------—-

   procedure Put_Trace (Info : String) is
   begin
      if Trace then
         Put_Line (Info);
      end if;
   end Put_Trace;

   ------------
   -- Trace_On –
   ------------

   procedure Trace_On is
      Id : System.Address;
   begin
      Id := Register_Thread;
      Trace := True;
      Unregister_Thread;
   end Trace_On;

   ------------—-
   -- Trace_Off –
   --—————-

   procedure Trace_Off is
      Id : System.Address;
   begin
      Id := Register_Thread;
      Trace := False;
      Unregister_Thread;
   end Trace_Off;

end Traces;

with Traces; use Traces;
procedure Increment is
   Value : Integer := 1;
begin
   for J in 1 .. 60 loop
      Value := Value + 1;
      Put_Trace (Integer'Image (Value));
      delay 1.0;
   end loop;
end Increment;
```

After having compiled and loaded the application on your target, spawn it and control the traces by calling `trace_on` and `trace_off` from the shell. You should see numbers displayed in the VxWorks console when the traces are on.

```
-> sp increment
task spawned: id = 3b0dcd0, name = s1u0
value = 61922512 = 0x3b0dcd0
-> trace_on
value = 0 = 0x0
-> trace_off
value = 0 = 0x0
```

B.10 Simulating Command Line Arguments for VxWorks

The GNAT implementation of `Ada.Command_Line` relies on the standard C symbols `argv` and `argc`. The model for invoking "programs" under VxWorks does not provide these symbols. The typical method for invoking a program under VxWorks is to call the `sp` function in order to spawn a thread in which to execute a designated function (in GNAT, this is the implicit main generated by gnatbind. `sp` provides the capability to push a variable number of arguments onto the stack when the function is invoked. But this does not work for the implicit Ada main, because it has no way of knowing how many arguments might be required. This eliminates the possibility of using `Ada.Command_Line`.

(Note that this section is only marginally relevant to VxWorks 653 and not applicable to VxWorks MILS, since applications on these platforms are usually invoked as part of system startup. The situation described here is only relevant there if the application is being downloaded to the Module OS / vxKernel partition, and is being directly spawned).

One way to solve this problem is to define symbols in the VxWorks environment, then import them into the Ada application. For example, we could define the following package that imports two symbols, one an int and the other a string:

```
with Interfaces.C.Strings;
use Interfaces.C.Strings;
package Args is
   -- Define and import a variable for each argument
   Int_Arg : Interfaces.C.Int;
   String_Arg : Chars_Ptr;
private
   pragma Import (C, Int_Arg, "intarg");
   pragma Import (C, String_Arg, "stringarg");
end Args;
```

An Ada unit could then use the two imported variables `Int_Arg` and `String_Arg` as follows:

```
with Args; use Args;
with Interfaces.C.Strings;
use Interfaces.C, Interfaces.C.Strings;
with Ada.Text_IO; use Ada.Text_IO;
procedure Argtest is
begin
   Put_Line (Int'Image (Int_Arg));
   Put_Line (Value (String_Arg));
end Argtest;
```

When invoking the application from the shell, one will then set the values to be imported, and spawn the application, as follows:

```
-> intarg=10
-> stringarg="Hello"
-> sp (argtest)
```

B.11 Using addr2line on VxWorks

For general information about addr2line, see *Getting Internal Debugging Information* in the *GNAT User's Guide*.

B.11.1 Differences between VxWorks and native platforms

Using addr2line on the VxWorks, where modules are often dynamically loaded, is a bit different than on native platforms. When dealing with dynamically loaded modules, one needs to determine the offset at which the module has been loaded. This allows addr2line to correlate the target addresses of the stack trace with the addresses in the load module.

On VxWorks, there are two ways a module can be located into target memory:

- it is dynamically linked and loaded
- it is statically linked with the kernel

In both cases, the addresses used in the module are different from the ones used within the target memory address space under VxWorks.

In contrast, on a native system, the executable has the same addresses in the object file and in memory, as virtual addresses are used. There is no need to collate the addresses on target with those in the object module.

As addr2line uses the addresses in the module, we need to adjust the addresses returned as a traceback at run-time so that they can be correctly interpreted by addr2line. To do this manually, we would follow a procedure like this:

A symbol that is always present in the module is determined in order to compute the offset between the addresses at run-time and the addresses in the module. Its address in the module is subtracted from its address in memory, and the computed delta is added to each address in the traceback.

B.11.2 Using <target>-vxaddr2line

Manually performing the computation just described is tedious, so a tool has been introduced to automate the process: <target>-vxaddr2line, where <target> is the name of target (e.g. powerpc-wrs-vxworks for VxWorks 5.x and 6.x; powerpc-wrs-vxworksae for VxWorks 653). All that needs to be done is to give the name of the module, the address of the symbol `adainit` in memory and the traceback values as parameters. The computation is then done automatically, and the result is transmitted to addr2line, which returns the symbolic traceback in the usual manner.

B.11.3 An example

The example session of this section is for powerpc-wrs-vxworks. Replace this target name by the target you are using. Consider the following simple source code :

```
1
2 procedure CE is
3
4    procedure Raise_Constraint_Error is
5    begin
6       raise Constraint_Error;
7    end;
```

```

   procedure Sub_2 (Should_Raise_CE : Boolean) is
   begin
     if Should_Raise_CE then
       Raise_Constraint_Error;
     end if;
   end;

   procedure Sub_1 is
   begin
     Sub_2 (Should_Raise_CE => True);
   end;
begin
   Sub_1;
end;
```

- Build the example with gnatmake, providing the -E binder argument in this case so that a raw backtrace is returned for the unhandled exception. -g is required in any case because addr2line uses debug information to perform the symbolic transcription.

  ```
  $ powerpc-wrs-vxworks-gnatmake -g ce -bargs -E
  ```

- Load and run the resulting module on the target board. It raises the expected unhandled exception and generates the associated raw backtrace:

  ```
  -> ld < ce
  Loading /ce |
  value = 591824 = 0x907d0
  -> sp ce
  task spawned: id = 1b8aae0, name = s2u0
  value = 28879584 = 0x1b8aae0
  ->
  Execution terminated by unhandled exception
  Exception name: CONSTRAINT_ERROR
  Message: ce.adb:6
  Call stack traceback locations:
  0x3b8394 0x3b83e0 0x3b8420 0x3b8458 0x3b82f0 0x19a184
  ```

  Now convert the backtrace into symbolic references...

- Determine the address of a reference symbol of the module (we use the address of `adainit`), which we obtain by calling the VxWorks `lkup` function:

  ```
  -> lkup "adainit"
  adainit                  0x003b81d0 text     (ce.exe)
  value = 0 = 0x0
  ```

- We now have the information needed to run `vxaddr2line`:

  ```
  powerpc-wrs-vxworks-vxaddr2line ce 0x003b81d0 0x3b8394 ... 0x19a184
                                  -- ---------- ---------------------
                           exec file  Ref.addr.  Run-Time backtrace
  ```

  This gives the following:

  ```
  000001C0 at .../ce.adb:6
  ```

```
0000020C at .../ce.adb:12
0000024C at .../ce.adb:18
00000284 at .../ce.adb:21
0000011C at .../b~ce.adb:88
```

[meaningless output for the last address]

The last line is not shown here because it designates stub code within VxWorks and is not part of the user's application.

Note that on VxWorks 653 and VxWorks MILS, the program name argument to vxaddr2line is the ".sm" file for the partition in which the application is executing.

B.12 Removal of Unused Code and Data

As for all elf platforms using 2.16.1 GNU binutils, GNAT for VxWorks now supports unused code and data elimination. For a complete description of this functionality, please refer to the GNAT User's Guide.

However, the use of this functionality need some extra care for VxWorks. In fact, GNAT for VxWorks performs by default partial linking on all VxWorks versions, except VxWorks 6 in RTP mode.

Because of this partial linking, the unused code and data elimination requires the use of `-e` / `--entry` ld option to correctly work. The usage of these options is also described in the GNAT User's Guide.

For example, in order to compile my_program.adb, you can use the following command line:

```
powerpc-wrs-vxworks-gnatmake my_program.adb -cargs -ffunction-sections \
  -fdata-sections -largs -Wl,--gc-sections -Wl,--entry=my_program
```

B.13 Debugging

In general, two debuggers are available for GNAT applications targeting VxWorks: the Workbench debugger and gdb (via GPS).

The Workbench debugger can be used for all targets supported by Workbench, except possibly VxWorks 5, which has not been tested.

gdb can be used for all supported versions of VxWorks except:

- VxWorks 6 RTPs
- VxWorks 6 SMP applications (kernel and RTP)
- VxWorks MILS applications

In the Workbench debugger, the names of Ada objects declared within packages must be entered in a particular format for the debugger to recognize them and display their values. Specifically, any dots in the package name must be replaced by two consecutive underscores and all lowercase letters must be used.

For example, a variable named Alpha declared in package Ownship would be referenced as `Ownship.Alpha` within Ada source code. Within the debugger data views, the variable must be referred to as `ownship__alpha` for the name to be recognized. Wind River is working to remove this restriction.

In addition, you must set an environment variable to enable viewing the values of Ada variables with these names. Specifically, set `DFW_ENABLE_ADA_SYMBOL_SEARCH` to 1. You need only set this environment variable once, as long as it does not become undefined later.

B.14 Frequently Asked Questions for VxWorks

- When I run my program twice on the board, it does not work. Why not?

  Ada programs generally require elaboration and finalization, so the compiler creates a wrapper procedure whose name is the same as the Ada name of the main subprogram, which invokes the elaboration and finalization routines before and after your program executes. But the static part of the elaboration is handled while loading the program itself, so if you launch it twice this part of the elaboration will not be performed the second time. This affects the proper elaboration of the GNAT run-time, and thus it is mandatory to reload your program before relaunching it.

- Can I load a collection of subprograms rather than a standalone program?

  It is possible to write Ada programs with multiple entry points that can be called from the VxWorks shell (or from a C or C++ application). To do this, generate an externally-callable Ada subsystem (see *Binding with Non-Ada Main Programs* in the *GNAT User's Guide*. If you use this method, you need to call `adainit` before calling any Ada entry point.

- When I use the `break exception` command, I get the message `"exception" is not a function`, why?

  You are not in the proper language mode. Issue the command:

  ```
  (gdb) set language ada
  ```

- When I load a large application from the VxWorks shell using the `ld` command, the load hangs and never finishes. How can I load large executables?

  This is a classic VxWorks problem when using the default `rsh` communication method. Using NFS instead should work. Use the `nfsShowMount` command to verify that your program is in a NFS mounted directory.

- When I load a large application from the debugger using the wtx target connection, the load never finishes, why?

  Make sure that the memory cache size parameter of the target server is large enough. (`target -m big_enough_size`, or Memory cache size box in GUI.) See *Tornado 1.01 API Programming Guide*, Section 3.6.2.

- When I spawn my program under the VxWorks shell, interactive input does not work, why?

  Only programs directly launched from the shell can have interactive input. For a program spawned with the `sp` or `taskSpawn` command, you need to have file redirection for input:

  ```
  ->    # here you can have interactive input
  -> main
  ->    # here you cannot
  -> sp main
  ->    # neither here
  -> taskSpawn("ess",100,0,8000000,main)
  ->    # but you can input from a file:
  -> taskSpawn("Bae",100,0,8000000,main) < input_file
  ```

- The `errno` of the task(s) of my Ada application is not null even though my application is running correctly. Is that normal?

  Yes. As explained in the *VxWorks OS Libraries API Reference* in the `errnoLib` section, *most VxWorks functions return ERROR when they detect an error, or NULL in the case of functions returning pointers. In general, they set an error status that describes the nature of the error*.

  There are a large number of calls to VxWorks functions in the Ada run-time. Whenever a system call returns a value indicating an error, the error status is set to a non-zero value. So despite the checking of the return value is to determine an appropriate action, `errno` can still be non-null.

  Resetting the error status in the Ada run-time each time a VxWorks function is called would add unnecessary system call overhead and would not avoid masking error status information that may appear in user code. So this approach would not help.

  It is a good practice not to rely on the error status value to detect errors. It should be only used to get more information on errors that have already been detected by checking the code returned by the VxWorks function. To be sure to get the error status corresponding to the error detected in the code, the use of the debugger is recommended.

Appendix C Tornado 2.x / VxWorks 5.x Topics

This chapter describes topics that are specific to GNAT for Tornado 2.x / VxWorks 5.x configurations.

C.1 Support for Software Floating Point on PowerPC Processors

The PowerPC 860 processor does not have hardware floating-point support. In order to build and run GNAT modules properly, you need to install and invoke software-emulated floating-point support as follows:

- Under the subdirectory '`lib\gcc\powerpc-wrs-vxworks\<gcc-version>`' of the GNAT installation:
  - Create a file '`ada_object_path`' containing the following line:
    ```
    rts-soft-float\adalib
    ```
  - Create a file '`ada_source_path`' containing the following line:
    ```
    rts-soft-float\adainclude
    ```
- When using the compiler, specify '`-msoft-float`' as a compiler and a linker option, e.g.:
  ```
  $powerpc-wrs-vxworks-gnatmake -msoft-float module -largs -msoft-float
  ```

C.2 Stack Overflow Checking on VxWorks 5.x

GNAT does not perform stack overflow checking by default. This means that if the main environment task or some other task exceeds the available stack space, then unpredictable behavior will occur.

To activate stack checking, compile all units with the gcc option '`-fstack-check`'. For example:

```
$ target-gcc -c -fstack-check package1.adb
```

Units compiled with this option will generate extra instructions to check that any use of the stack (for procedure calls or for declaring local variables in declare blocks) do not exceed the available stack space. If the space is exceeded, then a `Storage_Error` exception is raised.

For declared tasks, the stack size is always controlled by the size given in an applicable `Storage_Size` pragma (or is set to the default size if no pragma is used).

For the environment task, the stack size is defined by the `stackSize` argument when spawning the Ada program via the `taskSpawn` routine. If no stack size is specified, the stack size is set to the VxWorks default value.

The GNAT run time retrieves stack boundaries information from the VxWorks kernel using the `taskInfoGet` routine provided by the `taskShow` library.

Therefore, in order to have stack overflow checking working the `taskShow` library must be linked into the VxWorks kernel. This can be done using one of the following methods:

- Defining `INCLUDE_SHOW_ROUTINES` in `config.h` when using configuration header files
- Selecting `INCLUDE_TASK_SHOW` when using the Tornado project facility

C.3 Debugging Issues for VxWorks 5.x

The debugger can be launched directly from the Tornado environment or from `GPS`. It can also be used directly in text mode as shown below:

The command to run `GDB` in text mode is

```
$ target-gdb
```

where *target* is the name of target of the cross GNAT compiler. In contrast with native `gdb`, it is not useful to give the name of the program to debug on the command line. Before starting a debugging session, one needs to connect to the VxWorks-configured board and load the relocatable object produced by `gnatlink`. This can be achieved by the following commands:

```
(gdb) target wtx myboard
(gdb) load program
```

where `myboard` is the host name or IP number of the target board, and `wtx` is the name of debugging protocol used to communicate with the VxWorks board. When the debugger is launched directly from Tornado, the proper `target` command is automatically generated by the environment. Note that a target server for `myboard` is expected to be running before launching the debugger.

The GNAT debugger can be used for debugging multitasking programs in two different modes. A minimal understanding of these modes is necessary to use the debugger effectively. The two modes are:

- Monotask mode: attach to and debug a single task. This mode is equivalent to the capabilities offered by CrossWind. The debugger interacts with a single task, while not affecting other tasks (insofar as possible). This is the default mode.
- Multitask mode: The debugger has control over all Ada tasks in an application. It is possible to gather information about these tasks, and to switch from one to another within a single debugging session.

It is not advisable to switch between the two modes within a debugging session. A third mode called System mode is also available and can be used in place of the Multitask mode. Refer to the Tornado documentation for this.

Among the criteria for selecting the appropriate mode is the effect of task synchronization on the application's behavior. Debugging a tasking application affects the timing of the application; minimizing such effects may be critical in certain situations. The two modes have different effects: monotask mode only affects the attached task: others will run normally (if possible). Multitask mode stops all tasks at each breakpoint and restarts them on single-step, next, finish or continue; this may help avoid deadlocks in the presence of task synchronization, regardless of the inherent latency of stopping and restarting the tasks.

C.3.1 Using the Debugger in Monotask Mode

There are two ways to begin your debugging session:

- The program is already running on the board.

  The sequence of commands to use this mode is:

  - Launch GPS (possibly from the Tornado menu)

    If launching from the command line:

    ```
    gps --debug=myapp --debugger powerpc-wrs-vxworks-gdb \
        --target=myboard:wtx
    ```

    Once GPS has opened, connect to the target using the menu selection:

    `Debug ⇒ Initialize ⇒ <no main file>`

    Verify that the debugger has access to the debug information of both your program and the kernel. The Console window should have a message "`Looking for all loaded modules:`" followed by the names of the modules on the board and "`ok`". If you have some error messages here instead of "`ok`", the debugging session may not work as expected.

  - Attach to the desired task using

    `Debug ⇒ Attach...`

    This task is stopped by the debugger. Other tasks continue to operate normally (unless they are blocked by synchronization with the stopped task). The source window should display the source code location at which the task has been stopped. If the call stack display is enabled, it should reflect the stack of the attached task.

- The program hasn't been loaded to the board

  - Launch GPS (possibly from the Tornado menu)
  - Load your program to the board:

    If launching from the command line:

    ```
    gps --debug=myapp --debugger powerpc-wrs-vxworks-gdb \
        --target=myboard:wtx
    ```

    Once GPS has opened, connect to the target using the menu selection:

    `Debug ⇒ Initialize ⇒ myapp`

    GPS should display:

    ```
    Downloading your_program ...done.
    Reading symbols from your_program...expanding to full symbols...done.
    ```

  - Set breakpoints in your program.

    WARNING: the initial breakpoints must be set in code executed by the / environment main task of the application.

  - Run your program using one of the three methods below:
    - Click on the <Start/Continue> button in the GPS toolbar
    - Menu

      ```
      Debug ⇒ Run
      ```
    - Type into the GPS Debugger Console window

      ```
      (gdb) run your_program
      ```

- Whichever method you chose to start your debugging session, you can use the following commands at this point:
  - Browse sources and set breakpoints
  - Examine the call stack (Debug ⇒ Data ⇒ Call Stack)
  - Go "up" and "down" in the call stack ("up" & "down" buttons)
  - Examine data (Debug ⇒ Data ⇒ Display local variables, or any of the other methods for viewing data in GPS)
  - Continue/finish

  Next/step/finish will only work if the top frame in the call stack has debug information. This is almost never the case when first attaching to the task since the task is usually stopped by the attach operation in the GNAT run-time. You can verify which frames of the call stack have debug information by:

  ```
  Debug ⇒ Data ⇒ Call Stack
  <right Button> (contextual menu inside the call stack window)
   add "file location"
  ```

  If the current frame does not have a "file location", then there is no debug information for the frame. We strongly recommended that you set breakpoints in the source where debug information can be found and "continue" until a breakpoint is reached before using "next/step". Another convenient possibility is to use the "continue until" capability available from the contextual menu of the Source window.

  You can also examine the state of other tasks using

  ```
  Debug ⇒ Data ⇒ Tasks
  ```

  but you can't "switch" to another task by clicking on the elements of the task list. If you try to, you will get an error message in the GPS debugger console:

  ```
  "Task switching is not allowed when multi-tasks mode is not active"
  ```

  Once you have completed your debugging session on the attached task, you can detach from the task:

```
File ⇒ detach
```

The task resumes normal execution at this stage. WARNING: when you detach from a task, be sure that you are in a frame where there is debug information. Otherwise, the task won't resume properly. You can then start another attach/detach cycle if you wish.

Note that it is possible to launch several GPS sessions and simultaneously attach each to a distinct task in monotask mode:

```
Debug ⇒ Initialize ⇒ <no main file>
Debug ⇒ Attach...     (in the new window)
Debug ⇒ Detach
```

C.3.2 Using the Debugger in Multitask Mode

The steps are as follows

- Launch GPS (possibly from the Tornado menu)

  There are two possibilities:

  - If the program is already loaded on the target board, you need only verify that debug information has been found by the debugger as described above.
  - Otherwise, load the program on the board using

    ```
    Debug ⇒ Initialize ⇒ myprogram
    ```

- Set breakpoints in the desired parts of the program
- Start the program

  The simplest way to start the debugger in multitask mode is to use the menu

  ```
  Debug ⇒ Run
  ```

  and check the box `"enable vxWorks multi-tasks mode"`. You can also use the following gdb commands in the console window

  ```
  (gdb) set multi-tasks-mode on
  (gdb) run your_program
  ```

- Debug the stopped program

  Once stopped at a breakpoint (or if you interrupted the application), you can use all the standard commands listed for monotask mode + task switching (using Debug ⇒ Data ⇒ Tasks). Using next/step in this mode is possible with the same restrictions as for monotask mode, but is not recommended because all tasks are restarted, leading to the possibility that a different task hits a breakpoint before the stepping operation has completed. Such an occurrence can result in a confusing situation for both the user and the debugger. So we strongly suggest the use of only breakpoints and "continue" in this mode.

A final reminder: whatever the mode, whether you are debugging or not, the program has to be reloaded before each new execution, so that data initialized by the loader is set correctly. If you wish to restart the execution of a program, you can use the following sequence of gdb commands in the debugger console window:

```
(gdb) detach
(gdb) unload your_program(.exe)
(gdb) load your_program(.exe)
(gdb) run your_program
```

C.3.3 Debugging an Ada Application Spawned from a C Program

The previous sections have shown how to debug an Ada application that is either run from the debugger (can then be debugged in mono or multitask mode), or that is already running on the target (can be debugged in monotask mode). Now, you may be in a situation where the Ada application is spawned from a C program via `taskSpawn`. If you need to debug the application in monotask mode, then the previous section about a debugging session in monotask mode applies. But if you wish to debug the Ada application in multitask mode, a special procedure must be followed.

Let's assume that you are in the following situation: a C application (here called "loader") loads a multitasking Ada application from a disk or network, and spawns it using `taskSpawn`. An error appears in the Ada part and you need to debug it in multitask mode. Here are the steps to follow (the example code is provided at the end of the chapter):

- The target server and the VxWorks kernel must be configured to enable synchronization between the host and target symbol tables (see the Tornado/VxWorks documentation). This is required as the symbols of the Ada application are added to the target symbol table, while the debugger uses the target server (host) symbol table. If there is no synchronization between the two tables, the debugger will not have access to the symbols of the Ada application.
- In your C code, replace `taskSpawn` by `taskInit`. Like `taskSpawn`, `taskInit` will initialize the task. However, it will not start it. The initialized task will remain in *SUSPEND* mode.
- Once you have recompiled and loaded your C application, start it.
- Start the debugger. It should see the loaded Ada module.
- In the debugger console, set the debugger to multitask mode:

  ```
                (gdb) set multi-tasks-mode on
  ```
- Attach the debugger to the Ada task
- You can now set breakpoints and start your debugging session using the `continue` command, either from the gdb console or from the GUI.

Example of a C application ("loader") loading and initializing an Ada application (diners):

```
/*
 * This is an example on how to load and spawn an application from
 * a C program using two different methods: via taskSpawn, or by
 * using taskInit and taskActivate.
 *
 * With taskSpawn, the task will be initialized and run. A call
 * to taskInit followed by a call to taskActivate gives a similar
 * result.
 *
 * If taskInit is called alone, then the task is only initialized
 * and not started. The advantage of this approach is that you can
 * then attach a debugger to the task and start a debugging session.
 * If the spawned application is a multitasking application written
 * in Ada, then you can debug it in multitask mode.
 *
 * By default, taskSpawn is used. To use taskInit, you should define
 * the symbol DEBUG.
 *
 * Note that VX_FP_TASK is set when spawning the main Ada program.
 * This is required if the program uses any floating-point operations
 * at all, to ensure proper saving and restoring of floating-point
 * registers. The GNAT run-time takes care of setting VX_FP_TASK for
 * tasks other than the environment task, but you need to set this
 * flag explicitly for the environment task.
 */

#include <vxWorks.h>
#include <loadLib.h>
#include <taskLib.h>
#include <symLib.h>
#include <sysSymTbl.h>
#include <ioLib.h>
#include <stdio.h>

#define STACK_SIZE     0x100000

int loader (void) {
  int fd;
  int STATUS;

  char *module_name = "diners";
  MODULE_ID mod_id;
  FUNCPTR pValue;
  SYM_TYPE pType;

  char *tMonitorStack;

  /* allocate memory for the task stack and task control block */
  tMonitorStack =
    (char *) malloc (STACK_ROUND_UP (STACK_SIZE) + sizeof (WIND_TCB));

  /*
   * load the module to the target and determine its entry point
   */
  printf ("Opening the module...\n");
  fd = open (module_name, O_RDONLY, 0);
  if (fd == ERROR) {
    printf ("Could not open the module %s\n", module_name);
    return;
  }

  printf ("Loading module...\n");
  mod_id = loadModule (fd, LOAD_GLOBAL_SYMBOLS);
  if (mod_id == NULL) {
    printf ("ERROR: could not load the module\n");
    STATUS = close (fd);
    return;
  }

  STATUS = close (fd);

  printf ("Find the address associated to the symbol '%s'...\n",
          module_name);
  STATUS = symFindByName (sysSymTbl, module_name,
                          (char **)&pValue, &pType);
  if (STATUS == ERROR) {
    printf ("ERROR: could not find the symbol %s\n", module_name);
    return;
  }

#ifdef DEBUG
  printf ("Initializing the task...\n");
  STATUS = taskInit
    ((WIND_TCB *) (tMonitorStack),
     module_name,
     100,
     VX_FP_TASK,
#if (_STACK_DIR == _STACK_GROWS_DOWN)
     (char *) (tMonitorStack +
               STACK_ROUND_UP (STACK_SIZE) +
               sizeof (WIND_TCB)),
#else
     (char *) (tMonitorStack + sizeof (WIND_TCB)),
#endif
     STACK_SIZE,
     pValue, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  if (STATUS == ERROR) {
    printf ("ERROR: could not initialize the module\n");
    return;
  }

  printf ("done...\n");

  /*
   * Uncomment the following lines if you do not want the Ada application
   *  to remain in SUSPEND mode
   */
  /* printf ("Starting the task...\n");
     taskActivate ((int) tMonitorStack); */

#else
  printf ("Starting the task with taskSpawn...\n");
  STATUS = taskSpawn
    (module_name,
     100,
     VX_FP_TASK,
     STACK_SIZE,
     pValue, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  printf ("done...\n");
#endif
}
```

C.4 Using GNAT from the Tornado 2 Project Facility

This section describes how to add an Ada module to a Tornado project using the Tornado 2 Project facility described in *Tornado User's Guide*, Chapter 4. All instructions apply to both 'Downloadable Modules' and 'Kernel' project types.

C.4.1 GNAT as New Toolchain in the Tornado 2 Project Facility

Tornado 2 allows you to integrate third-party C toolchains. (*Tornado 2 API Programmer's Guide*, Chapter 7). Thus the GNAT toolchain will be seen as a new C toolchain when used from the Tornado 2 Project Facility. For each processor you have chosen during the GNAT/Tornado2 install, you have a corresponding <proc>gnat toolchain, e.g. PPC604gnat. These toolchains allow you to include Ada modules into your Tornado projects, and to build them directly using the Tornado build facilities.

The name of the "C compiler" in GNAT toolchains is *cc_gnat_<arch>*; the name of the 'linker' is *ld_gnat_<arch>*, where <arch> is an architecture (e.g. PPC). Associated build scripts call the appropriate executables during the build process: the C compiler, the C linker, or the GNAT toolchain, depending on the context.

When using Tornado, you can create two types of projects:

- A bootable VxWorks image
- A downloadable application

The integration applies to both cases, but it is only needed if you are building an Ada application. However, the integration (the GNAT toolchain and the BSPs modified for GNAT) can be used to build C applications as well. In this situation, you'll get the same behavior as when using corresponding toolchain (e.g. PPC604gnu).

C.4.2 Building a Simple Application

First, create a new project, using one of the GNAT toolchains.

To add an Ada source file to the current project, click on `Project` ⇒ `Add/Include`, browse to the relevant file, and include it. The selected Ada source file should contain the application's main subprogram. Any other required Ada source files will be compiled and linked by the underlying tools. Note that a Tornado project may contain only one Ada application, and therefore only one Ada main subprogram.

You can now build the project, using `Build` ⇒ `Rebuild all`. A log of the build process will be placed in the file '`gnatbuild.log`' in the build directory. It contains all calls made by the scripts, and information associated with each such tool invocation.

C.4.3 Mixing C and Ada Code in a Tornado 2 Project

When using the GNAT toolchain in the Tornado 2 Project Facility, you can mix C and Ada code in your project: you can have one main Ada program or no Ada program, and one or more C source files. For more information on mixed language programming, see *Mixed Language Programming* and *Interfacing to C*.

Please note that the GNAT/Tornado2 integration cannot currently handle mixed C / Ada applications, where the C code provides the main entry point. In this situation, use the manual build procedures given in the sections referenced above.

C.4.4 Compilation Switches

Once you have included all your source files, you may modify various build options. To pass options to the GNAT toolchain, go to the Project's build settings, on the `C/C++ Compiler` tab, and add your arguments into the input window.

Use the following rules to pass arguments to GNAT. The arguments should be:

- after any arguments passed to the C toolchain.
- prefixed with a switch identifying the tool that uses them:
  - `-margs` *gnatmake-options* to pass arguments to gnatmake
  - `-cargs` *gcc-options* to pass arguments to gcc
  - `-bargs` *gnatbind-options* to pass arguments to gnatbind

- `-largs` *gnatlink-options* to pass arguments to gnatlink

You can find more information on the compilation process of Ada source files in the section *The GNAT Compilation Model*. For a list of all available switches, refer to the sections describing `gnatmake`, `gnatbind` and `gnatlink`.

Here is an example that passes the option `-v` to gcc:

```
-g -mstrict-align -prjtype $(PRJ_TYPE) -ansi -nostdinc -DRW_MULTI_THREAD
-D_REENTRANT -fvolatile -fno-builtin -fno-for-scope -I.
-I/usr/windppc-2.0/target/h -DCPU=PPC604 -cargs -v
```

Here is an example that passes `-a` to gnatmake, `-gnatf` and `-gnatD` to gcc and `-E` to the binder:

```
-g -mstrict-align -prjtype $(PRJ_TYPE) -ansi -nostdinc -DRW_MULTI_THREAD
-D_REENTRANT -fvolatile -fno-builtin -fno-for-scope -I.
-I/usr/windppc-2.0/target/h -DCPU=PPC604 -margs -a -cargs -gnatf -gnatD
-bargs -E
```

In both examples, the switches before `-margs`, `-cargs`, `-bargs` and `-largs` are automatically added by the Tornado Project Facility. They are passed to the C compiler.

Note: The `-prjtype $(PRJ_TYPE)` option present in a few input boxes is used by the GNAT toolchain. It is required for the compilation process. You should not remove it from any input box.

C.4.5 Autoscale and Minimal Kernel Configuration

The Autoscale feature of the Tornado Project Facility can be used to determine the minimum set of kernel components required by your application when building a bootable VxWorks image. (Please refer to the *Tornado II User's Guide* Section 4.4 for more details). This feature is also available for projects involving Ada code. Just click on `Project⇒Autoscale` to launch a check to identify the minimal kernel configuration.

C.4.6 Adapting BSPs to the GNAT toolchain

To use a Board Support Package with the GNAT toolchain, it must be modified. This can be done manually, or by using the `setup` (Solaris) or `gnat-<gnat_version>-t2-integration-windows.exe` (Windows) programs provided in the GNAT/Tornado2 integration.

- To automatically create BSPs for the GNAT toolchain from existing ones:
  - Run `setup bsps` (on Windows, run the GUI-based integration tool)
  - Enter your Tornado install dir (i.e. the value of `WIND_BASE`)
  - Choose the BSPs from which new BSPs for the GNAT toolchain should be created.

Note : the BSPs detected are located in the directory `$(WIND_BASE)/target/config/`. The newly-generated BSPs are also placed in this directory. They have names of the form: `<BSP>_GNAT`.

This automatic generation and installation operates as described below to perform the adaptation.

- To adapt BSPs manually:

  The generic procedure is described in the *Tornado API Programmer's Guide*, Chapter 7 (this chapter is only available in Tornado 2.0.2 documentation).
  - Copy your BSP directory to a new directory
  - Go to this directory
  - Edit the file '`Makefile`',
    - Set TOOL to gnat: `TOOL=gnat`
    - Reverse the order of the following lines
      - `include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)`
      - `include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)`

C.4.7 Using GNAT Project Files in a Tornado 2 Project

You can use GNAT Project files to build your Ada applications. To do so, use the '`-Pproject_file`' option from `gnatmake`. The path to the project file can be either absolute or relative to the Tornado build directory, i.e. where the executable will be placed (e.g. '`~/myproject/PPC604gnat`').
When used in Tornado, the GNAT project file needs to be modified, as the module built by GNAT will be generated into the `Exec_Dir` specified by the project file. The `Exec_Dir` needs to be set to the Tornado build directory, so that Tornado can find the Ada load module. A variable (`TARGET_DIR`) is automatically set by the GNAT/Tornado integration to the Tornado build dir and given to the GNAT project. One then needs to retrieve this variable in the project to specify the `Exec_Dir`:

```
project Foo is

   Tornado_Exec_Dir := external ("TARGET_DIR");
   for Exec_Dir use Tornado_Exec_Dir;

end Foo;
```

This is the recommended method for using GNAT projects with the GNAT/Tornado integration.

One possible development scenario is to first build and test the Ada application on the host,and then move it to the target after initial checkout. The first part is done outside of Tornado with host tools; the second uses the GNAT/Tornado integration. A single GNAT project file can support both phases. The mechanism is to use an external variable to indicate whether the application is being built for host or target execution. The `Exec_Dir` (and other options) are specified to depend on the value of the external variable.

A typical simple GNAT project file looks like this:

```
project Foo is

   type Env_Type is ("Default", "Tornado");
   Env : Env_Type := external ("Env", "Default");

   Module_Dir := external ("TARGET_DIR", "./");

   for Main use ("hello.adb");

   case Env is

      when "Default" =>
         for Exec_Dir use "your_dir";

      when "Tornado" =>
         for Exec_Dir use Module_Dir;

   end case;

end Foo;
```

When working in the first phase of your project, one just needs to call:

```
gnatmake -Pfoo
```

Then in Tornado, the main file `hello.adb` needs to be added to the Tornado project, and the following compilation switches need to be passed to the GNAT toolchain (see Section C.4.4 [Compilation Switches], page 52):

```
-margs -P<dir_to_foo>/foo -XEnv=Tornado
```

Now, the Ada application can be built, and loaded or linked to the VxWorks kernel from Tornado, and will use the specifications in the GNAT project file.

C.5 Using GNAT with the RTI ScopeTools

This section applies to GNAT versions 3.16a1 and 5.01a and higher, and to Tornado 2.2 / VxWorks 5.5 with ScopeTools 4.0.

You can use GNAT with the following RTI ScopeTools:

MemScope
The instant memory analyzer

ProfileScope
The statistical profiler

TraceScope
The execution-flow trace tool

StethoScope
    The real-time data monitor

The code coverage analysis tool CoverageScope is not designed to be used with Ada. This is due to the underlying technology that parses and modifies the application source file. The parser is limited to C and C++.

C.5.1 General Information on Using the Tools

Since GNAT and Wind River use the same compiler technology (GCC), the objects generated by both compilers for C and Ada are similar, and GNAT integrates smoothly with the RTI tools.

C.5.1.1 Debug Switch

The RTI tools are designed to work with modules compiled with DWARF2 support. By default, the GNAT compiler generates STABS. To use the ScopeTools with GNAT, you should therefore replace the '`-g`' switch by '`-gdwarf-2`' that causes DWARF2 debugging information to be generated:

```
$ powerpc-wrs-vxworks-gnatmake -gdwarf-2 hello.adb
```

This is especially useful if you want to use the "goto-source" feature available in the RTI ScopeTools.

C.5.1.2 Name Demangling

Ada names, like C++ names, need "demangling". Although Ada name demangling is not directly supported in the RTI ScopeTools, this is not an issue since, in most cases, the demangling is straightforward. Generally "__" (two consecutive underscores) in a variable name corresponds to "." in the mangled Ada name.

Consider the following package:

```
package Foo is

   procedure My_Proc;

end Foo;

package body Foo is

   procedure My_Proc is
   begin
      null;
   end My_Proc;

end Foo;
```

If the procedure `My_Proc` is referenced somewhere in one of the RTI tools, it will appear with the name `foo__my_proc`, from which the corresponding Ada name `Foo.My_Proc` can be derived.

For more complex situations, the comments in the GNAT run-time file '`exp_dbug.ads`' supplies details.

C.5.1.3 Symbol Downloading

Generally, you will download all the symbols (global and local) of your application when you use the Ada and the RTI tools; e.g.:

```
    -> ld 1,0,"foo"
```

Otherwise, you may get incomplete information when using ScopeTools.

If you see unexpected symbols like `.LMx`, this is a known issue in ScopeTools 4.0 when used with GNAT, and you should install a patch from the RTI web site that addresses this issue.

C.5.2 MemScope

MemScope works with Ada applications in the same way that it works with C applications, since dynamic allocation / deallocation in Ada are directly mapped to the corresponding C functions `malloc` and `free`.

There is a known issue with the tracing of PPC callbacks (you will get a MemScopeTracingError message). If you receive such an error message, you can get a repair patch on the RTI web site.

C.5.3 ProfileScope

ProfileScope works with Ada applications, but see the discussion above regarding the download of global symbols.

C.5.4 TraceScope

TraceScope works with Ada applications. The Ada functions / procedures are listed in the registration table, and you can activate the functions as tracepoints.

Return values for functions work for the types `Boolean` and `Integer`.

Parameter values work for the predefined types `Float`, `Integer`, `Character`, `Long_Float`, and `Long_Integer`, but not for `String`.

C.5.5 StethoScope

There are two ways to use StethoScope for monitoring an Ada variable:

- Use the Ada binding to the StethoScope API (see the file '`filescope.ads`' in '`$WIND_BASE/rti/scopetools.4.0c/src/ada`'), or
- Export the Ada variable to its C equivalent so that it can be recognized by StethoScope.

For example:

```
package Foo is

   My_Var : Integer;
   pragma Export (C, My_Var, "my_var");

end Foo;
```

Then in the automatic signal installation, you should instruct StethoScope to monitor the variable `my_var`.

At present you have to use a pragma Export. Any attempt to directly access `My_Var` by giving StethoScope the Ada demangled name (i.e. `foo__my_var`) will lead to a “Signal Installation Error”.

Appendix D Workbench / VxWorks 6.x Topics

This Appendix describes topics that are specific to VxWorks 6. It introduces the basic features provided by the GNAT toolchain for this target, and shows how to use them. It also describes which functionalities are implemented, and summarizes known issues.

D.1 GNAT for VxWorks 6 Installation

The packages shipped with GNAT for VxWorks 6 have several installation dependencies that you need to understand in order to build a user-friendly setup:

- *GNATbench:* this provides build functionalities for both VxSim and PowerPC targets. All build targets whose name starts with PPC, PENTIUM, SIMPENTIUM, SIMLINUX or SIMNT are supported.

To sum up, a proper installation order is:

- Workbench;
- VxWorks 6;
- the core compilation system package;
- GPS;
- GNATbench;

D.2 Using GNAT for VxWorks 6

GNAT for VxWorks 6 comes with a choice of two development environments:

- GNAT Programming Studio (GPS), which is the standard GNAT IDE,
- GNATbench, the GNAT plug-in for Workbench (Wind River's IDE based on Eclipse).

Both environments share a common core technology, the GPS engine, so the choice of environment is mostly a matter of taste. GPS is more GNAT-centric and Workbench with the GNATbench plug-in is more VxWorks-centric.

The debugger technology depends on the environment. The underlying debugger is gdb in GPS, and DFW in Workbench. Both debuggers are well integrated into their respective development environments. The debugging interface in Workbench has a more VxWorks-centric view, and has practical displays to interact with the target. GPS has, via gdb, additional knowledge about Ada-specific data structures and provides additional facilities to display Ada variables.

The choice between the GPS and the Workbench development environment will be determined by your development habits and your personal preferences.

Note that, when using Workbench, you can easily switch to GPS to use some GPS-specific features such as the entity viewer or the dependency graph, and then switch back to Workbench. GPS is opened with the same files as Workbench, and file synchronization is assured between GPS and Workbench at each switch time.

D.3 Building a VxWorks 6 Application

GNAT for VxWorks 6 supports two build interfaces:

- Workbench;
- GPS;

for three types of modules:

- downloadable kernel modules (DKM)
- real time process (RTP) modules
- statically linked kernel modules (SKM) for VxWorks Cert

D.3.1 Building from Workbench

The GNATbench Ada Development User Guide describes how to use GNATbench within Workbench. This document, like all Workbench user guides, is accessible via the `"Help Contents"` sub-menu entry under the top-level Workbench `"Help"` menu.

See the `"Building"` chapter for all the topics related to building Ada projects within Workbench. This chapter includes general material, such as an overview of the predefined and GNATbench-defined build commands, as well as specific information that is essential to building with GNATbench.

You should also note that GNATbench provides a number of tutorials that provide detailed, step-by-step instructions and screen-shots showing how to create and build VxWorks projects. In addition to General Purpose Platform (GPP) projects, there are also tutorials on creating and building projects not yet directly supported by GNATbench. All these tutorials are located under the top-level node of the GNATbench Ada Development User Guide.

D.3.2 Building from GPS

Chapter `"Working in a Cross Environment"` in the *GPS User's Guide* explains how to adapt the project and configure GPS to work in a cross environment.

D.3.3 RTPs and kernel modules

To support both RTPs and kernel modules, two different run-time libraries are provided. For example, you can build an Ada kernel module from the `demo1` example using the following command:

```
powerpc-wrs-vxworks-gnatmake -g --RTS=kernel demo1.adb
```

This will build a relocatable object that you can load in the kernel context. The '-g' switch adds debugging information to the module.

Stack overflow checking using the '-fstack-check' switch is supported for both kernel applications and RTPs. However, if this facility is to be used for kernel applications, the kernel must be built with INCLUDE_PROTECT_TASK_STACK enabled.

To build it as an Ada RTP module, you need to use the rtp run-time library. The compiler will automatically find the VxWorks 6 RTP libraries using the WIND_BASE environment variable. Note that if you want to build an application for Wind River Systems' VxSim simulator you will need to pass '-vxsim' option to the linker. Otherwise the application will be built for the real x86 target. The previous example may be built as follows:

```
powerpc-wrs-vxworks-gnatmake -g --RTS=rtp -mrtp demo1.adb -largs -Wl,-L$WIND_BASE/target/usr/lib/ppc/
```

To do so in GPS, you need to update your project properties and add the options to the corresponding project attributes (e.g., '--RTS=rtp', '-mrtp' for Ada make options).

Note that use of gprconfig/gprbuild will insert the necessary includes, library references and '-mrtp' switches for compilation and linking automatically.

If you are storing a RTP in a ROM filesystem (for example when using RTPs with the VxWorks Cert 6.x certified kernel), you must specify the base address for the RTP using a linker switch. This one generally works due to default memory layout:

```
package Linker is
  for Default_Switches (‘‘Ada’’) use (..., "-Wl,--defsym,__wrs_rtp_base=0x40000000", ...);
end Linker;
```

Finally, if you are using the rtp-smp run-time library, you will want to add '--LINK=cc<arch>' or '--LINK=c++<arch>', where <arch> is an architecture (e.g. ppc or pentium), into the package Linker Default_Switches attribute. None of the above special switches need to be provided when using Workbench with the GNATbench plugin to perform builds, except for defining __wrs_rtp_base for VxWorks RTPs residing in the ROM filesystem.

D.4 SMP Support

Starting with VxWorks 6.6, the OS supports symmetric multi-processing (SMP) for specific target boards. GNAT includes run-time libraries supporting DKM and RTP applications running on a SMP kernel.

To use these libraries, use '--RTS=kernel-smp' or '--RTS=rtp-smp', depending on the application type.

When using these libraries, the VxWorks kernel must be configured with __thread support.

These libraries can also be used for uniprocessor kernels starting with VxWorks 6.6.

Processor affinity is controlled by using `pragma Task_Info`, by declaring a discriminant of type `System.Task_Info.Task_Info_Type` for a task type and giving it a value on task creation, or by using Ada 2012 `pragma CPU`.

For the first two options, allowable values range from 0 to n-1, where n is the number of processors available. A value of zero sets the processor affinity to VxWorks logical processor 0, and so on. For `pragma CPU`, allowable values range from 1 to n, and the affinity in VxWorks terms is set to the logical processor with an ID one less than the specified value. Out of range values, or an attempt to use this mechanism on VxWorks versions prior to 6.6, will cause creation of the task to fail.

By default, when the above mechanisms are not used, tasks inherit the processor affinity of the task that creates them (but see the notes on RTPs in the VxWorks user manuals).

The Workbench debugger should be used to debug SMP applications, as multiprocessor synchronization is not provided in gdb.

The rtp-smp run-time library uses the SJLJ exception handling mechanism, as does the kernel-smp library prior to version 6.5.x, which uses the ZCX scheme.

D.5 Using the VxWorks 6 Simulator

If you have access to the build toolchain for the simulator, it should be invoked using the `i586-wrs-vxworks` prefix. This toolchain also supports both RTP and kernel modules. The toolchain for the simulator is used in the same way as toolchains for real targets. Please see Section D.3 [Building a VxWorks 6 Application], page 60.

D.6 Debugging an Application on VxWorks 6

In VxWorks 6, the debugging interface used by the debuggers is a GDB/MI-based protocol named DFW. The component that offers debug services is called the DFW server.

Therefore, to be able to connect to the target board, you need first to have a DFW server running and your target board registered into this server. You can do that in Workbench using the panels `Remotes Systems` or `Target Manager`.

Note also that, to be able to run an RTP, you need access to the file system where the RTP module is located. This can be done in several ways, for example via FTP or NFS.

If the kernel has been loaded on the target board using an FTP server, the target board has access to the file system of the this server. Then, if your RTP module is accessible in the FTP server's file system, the target board will be able to load it.

If not, you will have to use one of the other file system clients that VxWorks provides. Here is an example for NFS. Assuming that you want to execute `/home/workspace/demo1.exe`, located on a machine named `rtphost` whose IP address is 192.168.0.1 and which has a mountable NFS file system, you can execute it from the target shell with the following commands:

```
hostAdd ("rtphost", "192.168.0.1")
nfsMount ("rtphost", "/home/workspace", "/mnt")
```

In any case, in the target server properties it is important to set the Pathname Prefix Mapping (corresponding to the previous settings) when creating the target server (section 19.1.3 of the *Wind River WorkBench User's Guide*):

```
Target Path       Host Path
rtphost:/mnt   /home/workspace
/mnt          /home/workspace
```

After this preliminary setup, you are ready to debug. A debugging session on VxWorks 6 can be divided into the following steps:

- connect to the target board;
- load the module on the target memory;
- execute and debug your program.

The following sections explain how these three steps map into the two provided debugging solutions: Workbench and GPS.

D.6.1 Using Workbench DFW

GNATbench is integrated smoothly into Workbench's debugging solution. It provides Ada support similar to the what is available for C (breakpoints, control of the debugger execution, etc.). Note that a few Ada types use a debug encoding that Workbench cannot read yet. In that case, Ada variables using these types are not displayed by Workbench. A workaround is to use GPS for debugging.

D.6.2 Using GPS and GDB

As mentioned previously, the Ada debugger in GPS uses the DFW interface for controlling the execution on the program executing on the target. To be able to debug an Ada program, the DFW server has to be launched from Workbench and has to be connected to the target you are debugging.

Note that gdb can only debug uniprocessor kernel applications; the Workbench debugger should be used for RTPs and SMP applications.

Several DFW servers may have been launched on your host; if so, you will need to tell the debugger which one it should pick. To do so:

- In Workbench, open `Windows -> Preferences`, then expand `Wind River -> Debug Server Setting`. There, you should find the session name of the DFW server; e.g. `dfw-wb3111-${user}`.
- Set the environment variable `DFW_SERVER_NAME` to this name; e.g., on Unix: `DFW_SERVER_NAME=dfw-wb3111-me; export DFW_SERVER_NAME`

A kernel module generated by the GNAT toolchain can be loaded and run in the same way as a VxWorks 5.* kernel module (Ada or C). An example may be found in Appendix B [Common VxWorks Topics], page 15.

Before starting a debugging session in GPS, you should ensure that several fields in your GNAT Project are filled in:

- Program host: <name of your connection to the board>
- Protocol: `dfw` to debug an kernel task
- Debugger: `powerpc-wrs-vxworks6-gdb`

The name of your connection to the board is displayed in Workbench, in panel `Remotes Systems` or `Target Manager`, at the root of your target's connection tree.

As was mentioned previously, the three steps to debugging on VxWorks 6 are *connecting*, *loading*, and *executing*.

To connect to your target and debug a kernel module, the debugger uses a target interface named `dfw`. To do so in GPS, you can specify the target protocol in your project properties, and use the menu `Debug -> Initialize -> <no_main_file>`. Alternatively, if you want to use the command line debugger, you should use the `target` command:

```
(gdb) target dfw <myboard>
Connecting to DFW server <myhost>:1603...Connecting to <myboard>@<myhost>...
done.
```

where `<myboard>` is the name of the target board you are attaching to.

You cam load your module either during initialization in GPS, with one of the choices given by the menu (e.g., `Debug -> Initialize -> demo1`), or from the console:

```
(gdb) load demo1
```

You can then start executing, using the menu `Debug -> Run` or the `Start/Continue` button. As always, there is a corresponding call in the command line debugger, which is named `start` and takes the name of the main procedure as a parameter:

```
(gdb) start demo1
Breakpoint 1 at 0xad5768: file demo1.adb, line 4.
Starting program:  demo1
demo1 () at demo1.adb:4
4       procedure Demo1 is
```

Note that, if your module has already been loaded and executed from an other tool (e.g. Workbench or the host shell) you can also attach to the running process in the debugger. To do so, first get the list of kernel tasks and RTPs with the command `info vxworks-tasks`:

```
(gdb) info vxworks-tasks
0 task 0x604df868       (tShell0)
1 task 0x6037bc70       (tWdbTask)
2 task 0x6036f6d8       (ipnetd)
3 task 0x604159e0       (ipcom_syslogd)
4 task 0x60427af0       (tNet0)
5 task 0x60403a38       (tAioIoTask0)
6 task 0x60403740       (tAioIoTask1)
7 task 0x603e59a8       (tAioWait)
8 task 0x6037e290       (tNbioLog)
9 task 0x60387c38       (tLogTask)
10 task 0x60170774      (tExcTask)
11 task 0x6037a480      (tJobTask)
12 task 0x73acc0        (tMy_p)
```

This gives you the id of the different tasks. To debug a task, use the `attach` command. For example, to attach `tMy_p`:

```
(gdb) attach 0x73acc0
```

You can then set breakpoints, start and stop the execution, and display your variables... For more details, please refer to the *GDB User's Manual*.

Appendix E VxWorks 653 Topics

This Appendix provides information specific to the GNAT cross-development system for the VxWorks 653 target. Supported versions are selected VxWorks 653 1.8.x, 2.2.x and 2.3.x editions. See the release notes for details.

E.1 Introduction

VxWorks 653 is a time- and space-partitioned real-time operating system that conforms to the ARINC-653 standard. Its purpose is to support the implementation of Integrated Modular Avionics (IMA) system architectures and similar architectures in other industries. Because an application running in one partition cannot be affected by those in different partitions, applications can be developed with independently, and their safety certification can be unlinked from the safety certification of other subsystems.

The VxWorks 653 architecture and programming model is described in the *VxWorks 653 Programmer's Guide*.

For programming purposes the operating system (OS) is divided into two parts:

- The *Module OS*, which contains drivers and other privileged software. This part of the system presents an API that is described in the *VxWorks 653 Module OS API Reference*, *VxWorks 653 Module OS Errno Code List* and the *VxWorks 653 Programmer's Guide*.
- The *Partition OS*, which is used within application partitions. This part of the system presents an API similar to that of VxWorks 5.x, or the VxWorks 6.x kernel, called vThreads, and is also enhanced with optional POSIX and APEX (ARINC-653) components. It is primarily described in the *VxWorks 653 Partition OS API Reference*, *VxWorks 653 Partition OS Errno Code List* and the *VxWorks 653 Programmer's Guide*.

GNAT for VxWorks 653 lets you develop applications for the various VxWorks 653 partition types and operating modes:

- Module OS applications can be built with the "zero footprint" Ada run-time library (`rts-zfp`).

  Applications written for the Module OS may only use certain port components that must be imported by the application, as opposed to the full APEX bindings.
- Partition OS applications can be built with the full Ada run-time library (`rts-full`), the restricted Ravenscar run-time library (`rts-ravenscar-cert`), the restricted certifiable run-time library (`rts-cert`) or the zero footprint run-time library (`rts-zfp`). Application partitions may include

either the full or partial APEX components, or just the vThreads components. Ada tasking applications must use only the APEX_MINIMAL subset of APEX.

It is expected that nearly all applications developed with GNAT will execute in application partitions under the Partition OS.

This Appendix describes the mechanics of building and debugging Ada applications for VxWorks 653, and also treats other issues relevant to application development in the various contexts listed above.

E.2 Setting Up a VxWorks 653 System

E.2.1 The GNAT VxWorks 653 Starter Kit

A starter kit is available to ease the creation of a first VxWorks 653 system containing Ada, C or C++ applications. Information about the starter kit is also available as a '`README`' file in the GNAT VxWorks 653 Starter Kit package.

The starter kit provides a command line tool `vx653_system_build` that will create and build a full VxWorks 653 system.

The resulting system contains all the elements needed to start developing an Ada application on VxWorks 653. The system is configured with the following characteristics (a non-exhaustive list):

- two application partitions
- user-defined ports to provide inter-partition communication
- two schedules:
  - schedule0 (the default) allocates no time to the application partitions, so that a debugger may be attached.
  - schedule1 provides appropriate time allocations to allow the application partitions to execute.
- a Health Monitor with most entries set to `CFG_NO_HANDLER`. This is a very basic configuration that prevents any conflict between the Health Monitor and Ada exceptions.

Run `vx653_system_build` with no arguments to get command line help. `vx653_system_build` requires at least one parameter: the name of the configuration file to use. Some examples with detailed documentation are provided with the Starter Kit.

E.2.1.1 Configuration and Compilation Issues

The GNAT VxWorks 653 Starter Kit can be used in one of two possible modes:

- one of the default Ada/C systems provided by the starter kit is linked to the VxWorks kernel, or

- an external Ada/C/C++ system can be specified

Using '`Makefile.example`' as the configuration file, the Starter Kit will build the arinc653 example application, which gives a good overview of what should be done to get started with a VxWorks 653 application.

Using '`Makefile.testsuite`' as the configuration file, the Starter Kit will build a testsuite of the Ada binding to the ARINC653 APEX, and of other run-time library implementation features.

Both applications use Ada and C APEX processes, and use two application partitions to run.

In the second case, the user can specify which applications to run in the first and second partitions. Since the configuration file specifies entry points for the custom applications (FIRST_APP_MAIN, SECOND_APP_MAIN), the VxWorks 653 system will be configured so that the custom applications are started automatically when the partitions containing it are scheduled.

To build a system with only one partition, do not specify a location for the second partition (SECOND_APP_DIR). The resulting system will effectively have two partitions, but the second partition will just run a null program.

To use a custom application, these rules must be followed:

- a GNAT Project File must be used for each application partition. These files can handle, Ada, C and mixed-language applications
- this GNAT Project File should be in the root directory of your application's directory tree
- the resulting module should be put in the root directory of your application's directory tree

In a nutshell, the GNAT Project file and the executable should be in the same directory. See the file '`Makefile.example`' provided with the Starter Kit for more information on how to create a configuration file to use a custom application.

For VxWorks 653 2.2.x, the Starter Kit will generate the Workbench project files '`Makefile.wr`', '`.project`' and '`.wrproject`' are generated into the designated system installation directory (see INSTALL_DIR in '`Makefile.example`'). That directory and the project it contains can be imported into Workbench as an existing project when using Workbench 2.6.1 or 3.0 and GNATbench 2.0.1. Build targets will be created for various parts of the system and for the entire system. See the GNATbench help (in Workbench) for more information on using Workbench with VxWorks 653 Ada applications.

For VxWorks 653 2.2.x, you can also clean and rebuild the project or its parts as created by the starter kit by changing to the system installation directory and executing a command of the form:

```
make -f Makefile.wr [target]
```

from within the VxWorks 653 2.2 development shell.

For VxWorks 653 2.3.x (Workbench 3.1+), we recommend use of the latest version of GNATbench to create systems directly within Workbench as a better integrated and much more flexible alternative to using the starter kit. See the documentation installed with GNATbench.

E.2.1.2 Running the System

Once `vx653_system_build` has successfully created and compiled the system, the system can be directly loaded and run on the target. The compiled system is located in:

```
INSTALL_DIR/vxWorks
```

where INSTALL_DIR is the system installation directory specified in the configuration file.

When the system is started, the application partitions will not execute because the default schedule (`schedule0`) does not allocate any time to them. Use `schedule1` to start the two application partitions.

To specify the schedule to be used, type the following in either the target shell or the VxWorks shell:

```
[vxkernel] -> arincSchedSet (xxx, 0)
```

where *xxx* is the schedule number: `0` for `schedule0`; `1` for `schedule1`; etc.

E.2.2 Manually Configuring a VxWorks 653 System

The GNAT VxWorks 653 Starter Kit automates a number of steps that are required to include an Ada application in a VxWorks 653 system. The operations can be split into two parts: configuration to define a working VxWorks 653 platform, followed by configuration to add applications to the system.

Normally, on a real program, the platform will be jointly defined by specialists such as the platform developer and systems integrator. Application developers will then be provided with the platform and a description in XML configuration files of the resources available to them in the partitions their applications will execute. Definition of the platform is outside the scope of this manual, and is described in the *VxWorks 653 Configuration and Build Guide*.

In this section, we will focus on integration of applications into the platform. We suggest you first familiarize yourself with the overall process as described in the above reference.

E.2.2.1 Application Configuration Files

For each application, there are two sections of XML configuration info to be provided.

The first is in the partition description, which will be in either the module XML file, or in a partition-specific XML file. Refer to the *VxWorks 653 Config-*

uration and Build Guide for specific contents. An example fragment for a two partition system would be:

```
        <Applications>
             <Application Name="part1">
             <xi:include href="C:/testbench/first_app/config/application.xml"
    />
              </Application>
             <Application Name="part2">
    <xi:include href="C:/testbench/second_app/config/application.xml"
    />
              </Application>
        </Applications>

    ...

        <Partitions>
              <Partition Name="part1" Id="1">
                <PartitionDescription>
                    <Application NameRef="part1"/>
                    <SharedLibraryRegion NameRef="ssl"/>
                    <Settings
                              RequiredMemorySize="0x500000"
                              PartitionHMTable="part1Hm"
                              watchDogDuration="0"
                              allocDisable="false"
                              numStackGuardPages="0xffffffff"
                              numWorkerTasks="0"
                              isrStackSize="0xffffffff"
                              selSvrQSize="0xffffffff"
                              maxEventQStallDuration="INFINITE_TIME"
                              fpExcEnable="true"
                              syscallPermissions="0xffffffff"
                              numFiles="0xffffffff"
                              maxGlobalFDs="10"
                              numDrivers="0xffffffff"
                              numLogMsgs="0xffffffff"/>
                </PartitionDescription>
            </Partition>
               <Partition Name="part2" Id="2">
                <PartitionDescription>
                    <Application NameRef="part2"/>
                    <SharedLibraryRegion NameRef="ssl"/>
                    <Settings
                              RequiredMemorySize="0x400000"
                              PartitionHMTable="part1Hm"
                              watchDogDuration="0"
                              allocDisable="false"
                              numStackGuardPages="0xffffffff"
                              numWorkerTasks="0"
                              isrStackSize="0xffffffff"
                              selSvrQSize="0xffffffff"
                              maxEventQStallDuration="INFINITE_TIME"
                              fpExcEnable="true"
                              syscallPermissions="0xffffffff"
                              numFiles="0xffffffff"
                              maxGlobalFDs="10"
                              numDrivers="0xffffffff"
                              numLogMsgs="0xffffffff"/>
                 </PartitionDescription>
             </Partition>
        </Partitions>
```

The application part references external XML files used to define the application characteristics, and the applications defined in that fragment are then referred to in the partition fragments.

A typical application XML file will look like this:

```
<ApplicationDescription
xmlns="http://www.windriver.com/vxWorks653/ConfigRecord"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.windriver.com/vxWorks653/ConfigRecord Application.xsd">
<MemorySize
                        MemorySizeHeap="0x100000"
                        MemorySizeBss="0x80000"
                        MemorySizeText="0x100000"
                        MemorySizeData="0x100000"
                        MemorySizeRoData="0x20000"
                        MemorySizePersistentData="0x10000"
                        MemorySizePersistentBss="0x10000">
                        <AdditionalSection
Name=".gcc_except_table"
Size="0x10000"
Type="DATA"/>
</MemorySize>
  <Ports>
                    <QueuingPort
                              Name="queuing_answer_dest"
                              Direction="DESTINATION"
                              MessageSize="4"
                              QueueLength="100"
                              Protocol="NOT_APPLICABLE"/>
                    <QueuingPort
                              Name="queuing_sender"
                              Direction="SOURCE"
                              MessageSize="4"
                              QueueLength="100"
                              Protocol="SENDER_BLOCK"/>
                     <SamplingPort
```

```
                                   Name="sampling_sender"
                                   Direction="SOURCE"
                                   MessageSize="4"
                                   RefreshRate="0.001"/>
                     <SamplingPort
                                   Name="sampling_answer_dest"
                                   Direction="DESTINATION"
                                   MessageSize="4"
                                   RefreshRate="0.001"/>
</Ports>
</ApplicationDescription>
```

Its parts include a standard header that points to the corresponding schema, definition of the application entry point, the location of the generated object module and its name, memory area sizes, and an additional section to support C++ exception handling (if used). Finally, a section is provided that defines the APEX ports used by the applications. These are connected among partitions in the module XML file.

A final configuration file that you may need to create (it will usually be provided as part of the platform) defines the interface to the system shared library that contains the partition operating system. An example for the certified vThreads POS would contain something like the following, and would be used as part of the application partition build as described in the next section:

```
<Shared_Library_API
    xmlns="http://www.windriver.com/vxWorks653/SharedLibraryAPI"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    Name="vThreads"
    >

    <Interface>
<Version Name="Cert"/>
<xi:include
href="$(WIND_BASE)/target/vThreads/config/comps/xml/apex.xml" />
<xi:include
href="$(WIND_BASE)/target/vThreads/config/comps/xml/vthreads_cert.xml" />
    </Interface>
</Shared_Library_API>
```

Here we again see a standard header, followed by references to the interface description XML files for the components we need in the partition operating system.

If, as recommended, you compose your systems directly with Workbench 3.1+ using GNATbench for VxWorks 653 2.3+, you will need to provide full XML files rather than the fragments described here.

E.2.2.2 Partition Makefiles

Typical partition makefiles for C and C++ are described in the *VxWorks 653 Configuration and Build Guide*. For Ada and mixed applications, we use the GPRbuild tool, rather than using .o files as targets as is shown in the guide. Here is an example makefile for a partition containing an Ada or mixed language application. It is invoked from '`Makefile.wr`' in the usual way for building partitions, from within the VxWorks 653 development shell or Workbench.

Note that if you are using GNATbench with VxWorks 653 2.3.x or more recent, GNATbench will automatically generate the partition Makefile, so this section can be skipped.

```
# Makefile.part1 - makefile for partition 1

########################
#
# Parameters
#

# INSTALL_DIR: root of build location
# FIRST_APP_DIR, FIRST_APP_PROJ, FIRST_APP_MAIN: location, GPR file, module name
# SYS_CFG_DIR: directory containing ssl interface file ssl.xml
# XML_FILE: module XML file
# POS: cert, full
# RTS: cert, full, ravenscar-cert, zfp
# BINDING: APEX binding to use - apex, apex_95, apex_minimal, apex_zfp, apex_zfp_95
# CPU_TYPE: PPC, SIMNT

########################
#
# Variables
#

include $(WIND_BASE)/target/vThreads/config/make/Makefile.vars

USER_OBJ    = $(INSTALL_DIR)/part1/$(FIRST_APP_MAIN).o
USER_MODULE = $(INSTALL_DIR)/first_app/$(FIRST_APP_MAIN).out

CFLAGS_EXTRA = "-DUSER_APPL_INIT=$(FIRST_APP_MAIN)()"

ifeq ($(POS), cert)
   CFLAGS_EXTRA += -DCERT
endif

ifeq ($(POS),full)
   PART_OBJS    =  vThreadsCplusComponent.o \
       vThreadsCplusLibraryComponent.o \
   __ctype_tab.o
else
```

```
   PART_OBJS    =  vThreadsCplusComponent.o
endif

ifeq ($(CPU_TYPE), PPC)
   BUILD_SPEC = powerpc-wrs-vxworksae
else
   BUILD_SPEC = i586-wrs-vxworksae
endif

########################
#
# Rules
#
########################


include $(WIND_BASE)/target/vThreads/config/make/Makefile.rules

vpath %.gpr $(FIRST_APP_DIR)
vpath %.c   $(WIND_BASE)/target/vThreads/config/comps/src
vpath %.c   $(WIND_BASE)/target/vThreads/config/comps/src/templates

default: part1.sm part1.rpt

# ssl.xml is the interface definition, not the ssl definition:
ssl-stubs.c: $(SYS_CFG_DIR)/ssl.xml

FORCE_EXTERNAL_MAKE:

$(USER_MODULE): FORCE_EXTERNAL_MAKE
        gprconfig --batch --target=$(BUILD_SPEC) \
        --config=ada,,$(RTS) --config=c --config=c++
gprbuild -P \
        $(INSTALL_DIR)/first_app/$(FIRST_APP_PROJ) \
        -gargs -XRuntime=$(RTS) -XCPU=$(CPU_TYPE) -XBinding=$(BINDING)

$(USER_OBJ): $(USER_MODULE)
$(CP) $(USER_MODULE) $(USER_OBJ)


LDSFLAGS_EXTRA = -j $*
part1.lds: $(XML_FILE)

LDFLAGS_EXTRA = -T $*.lds

part1.sm: vxMain.o ssl-stubs.o $(USER_OBJ) $(PART_OBJS) part1.lds

clean:
$(RM) *-stubs.c *.o *.lds *.sm *.rpt
```

```
powerpc-wrs-vxworksae-gnatclean -P \
        $(INSTALL_DIR)/first_app/$(FIRST_APP_PROJ) \
        -XRuntime=$(RTS) -XCPU=$(CPU_TYPE) -XBinding=$(BINDING)
```

E.3 Running and Debugging Applications

VxWorks 653 1.8.x provides a C-oriented debugger that can be used in a limited way on Ada applications. VxWorks 653 2.x provides a more sophisticated debugger that can handle C, C++ and Ada. GNAT provides a VxWorks 653-targeted version of gdb that has been enhanced to provide Ada-knowledgeable debugging (as well as supporting C and C++). This version of gdb is normally invoked from within the GNAT Programming System (GPS) development environment. This section describes the use of the GPS debugger on VxWorks 653.

E.3.1 VxWorks 653 System Setup

In order to debug on VxWorks 653 you need to attach a debugger at the correct time during OS initialization. The suggested approach is to define two schedules in the system in the module XML file.

The first (default) schedule does not allocate any time to application partitions that we might wish to debug; the second gives the application partitions their expected allotments. For example:

```
<Schedules>
     <Schedule
              Id="0"
              Name="schedule0"
              MajorFrame="0"
              MinorFrame="0">
        <PartitionWindow
                         PartitionNameRef="part1"
                         Duration="0.0"
                         ReleasePoint="true"/>
        <PartitionWindow
                         PartitionNameRef="part2"
                         Duration="0.0"
                         ReleasePoint="true"/>
    </Schedule>
     <Schedule
              Id="1"
              Name="schedule1"
              MajorFrame="0"
              MinorFrame="0">
        <PartitionWindow
                         PartitionNameRef="part1"
                         Duration="0.0001"
                         ReleasePoint="true"/>
        <PartitionWindow
```

```
                                 PartitionNameRef="part2"
                                 Duration="0.0001"
                                 ReleasePoint="true"/>
            </Schedule>
        </Schedules>
```

This approach will ensure that none of the application partitions will start execution after the system boots. We can then attach a debugger to a partition before it starts, so we can debug it from its beginning.

E.3.2 Environment Setup

For PSC 1.8.x, it is necessary to set up the Tornado environment variables for the debugger before launching GPS. These variables are set by the `torVars` script. On UNIX machines, `source` the script that corresponds to the shell you are using – '`torvars.sh`' or '`torvars.csh`'. On a Windows machine, either make the system call the '`torVars.bat`' script on startup, or modify your environment variables manually so that you do not have to execute the script every time. For VxWorks 653 2.x execute GPS from within the VxWorks 653 Development Shell (`wrenv`) or from Workbench, as `torVars` is no longer used on these versions. Alternatively, you can debug using the Workbench debugger.

E.3.3 GPS Setup

In GPS, on the general properties page, designate the name of your target server in the `Program Host` field and select `wtx` as the protocol. Leave the `Tools host` field blank.

E.3.4 Debugging a Partition

The procedure to debug partition 1 is as follows:

1. Boot the target. It will eventually boot the partition OS and stop at a target shell prompt (assuming you've included a target shell). At this time, the Module OS has performed most of its initializations, and the partitions have been created, but not yet started.
2. Start your target server. This can be done from within Tornado, or from a command line (preferably using a shell script). It is recommended that you close the Tornado debugger if you start the target server from the Tornado GUI.
3. Within GPS, select `Debug⇒Initialize⇒<No Main Program>`
4. Within GPS, select `Debug⇒Attach`. A dialog will pop up with a list of all Module OS tasks. Select `tPartition1` from the list. The name given is constructed from the name of the partition as given in '`configRecord.c`', prefixed by `t`.
5. At this point, you should be able to set breakpoints within the application partition, including the one to receive control whenever an exception occurs.

You might also wish to display the call stack at this time by selecting `Debug⇒Data⇒Call Stack`.

6. Once you have set any desired breakpoints, hit the `Continue` button in GPS. The system is running, but since the first schedule is still being used by the core OS, the application partitions are not being scheduled.
7. Change to the schedule that allocates time to the application partitions by typing the following in the target shell:

```
                    arincSchedSet (1, 0)
```

You can now debug the partition as you would any other program. When your debugging session is finished we recommend you detach from your target (`Debug⇒Detach`) before closing the debugger.

E.3.5 Debugging Multiple Partitions

There are two methods that can be used to debug a multi-partition VxWorks 653 system. In both methods the Tornado debugger should not be running.

The first method is to use one GPS instance per partition. Each GPS instance should be attached to a partition, and no two instances should be attached to the same partition. Also, when debugging a multi-partition system you should not do system mode debugging with any of the instances of GPS.

The second method is to use system-mode debugging. In this mode, the debugger takes control of the entire system, and allows you to debug all partitions at the same time. To use that method, you need to type `"system"` as the name of the task to attach to. Only one debugger can be attached to the system.

E.3.6 System Mode Debugging

Note that `gdb` supports "system mode" debugging (see the Tornado documentation). To debug in system mode follow the steps provided above to start the debugger. Then open the task list dialog selecting `Debug⇒Attach`, type `SYSTEM` in the text entry at the bottom of the dialog, and click `OK`.

This will bring the target to system mode and attach the debugger to it. Breakpoints are preserved when switching between task mode and system mode.

Alternatively, you can start system mode debugging by typing the following in the gdb prompt:

```
(gdb) attach system
```

Note then that instead of using the `Continue button` to start your application, you will have to type the following in the gdb prompt:

```
(gdb) continue
```

To exit system mode you can use the `Debug⇒Detach` menu entry. This will detach the debugger from the target and resume the execution of your VxWorks 653 system.

E.3.7 Debugger Commands Specific to VxWorks 653

The debugger provides the following two commands related to the handling of partitions. These features are also available graphically in GPS, through the `Debug⇒Data⇒Protection Domain` menu entry.

`info pds`

This command prints the list of partitions available on the target. For each partition, the debugger prints its ID and name. An asterisk at the beginning of a partition entry identifies it as the current partition.

```
(gdb) info pds
   PD-ID        Name
*  0x15f78c     vxKernel
   0x8007a8     vxSysLib
   0x800890     pdsAppDomain
```

`pd` *`new-partition`*

This command switches the debugger to the new partition. The partition ID or the partition name can be used to identify the target partition.

Using the list of partitions from the example above, the two commands in the following example demonstrate how to switch to the vxSysLib partition using its partition ID (0x8007a8), and then how to switch to the pdsAppDomain using its name.

```
(gdb) pd 0x8007a8
[Switching to PD 0x8007a8 (vxSysLib)]
(gdb) pd pdsAppDomain
[Switching to PD 0x800890 (pdsAppDomain)]
```

E.4 Application Design Considerations

As a general comment, one should never mix APEX processes, POSIX threads or Ada tasks in a single application partition. Each of these entities is part of a distinct high-level model for developing concurrent applications; mixing them will lead to confusion and unpredictable application behavior.

Before starting, one should know what "foreign threads" are. "Foreign threads" refer to threads that are not created by the Ada run-time environment, and therefore are not known to it without taking additional bookkeeping steps. In order to facilitate seamless operation of such "foreign threads" when they execute Ada code, they need to be registered with the Ada run-time. On some native platforms, GNAT accomplishes this automatically; another alternative is to use GNAT.Threads (which can be used for raw vThreads on VxWorks 653).

Since VxWorks 653 applications are expected to comprise mainly APEX processes, the standard way to perform registration on VxWorks 653 is through the Apex_Processes.Create_Process routine in the Ada APEX binding provided with GNAT. By using this version of the binding, one can ensure that exception handling and other operations such as functions with unconstrained results and the process-level health-monitoring handler are executed correctly.

E.4.1 General Application Structure on VxWorks 653

For applications that are to execute in application partitions, there is a typical structure.

There is usually one "main" application that will perform initializations: allocating data, creating and starting various APEX processes, and finally setting the partition into "normal" mode. Once the partition has entered normal mode, the vThread executing the main is suspended, no more dynamic allocation is allowed, and the APEX processes defined within the partition are scheduled according to the characteristics defined for them within the main application.

For an Ada application, this means that the main subprogram is expected to perform initializations as described, and then to suspend until the partition is restarted. This suspension occurs when `Apex_Processes.Set_Partition_ Mode (Normal);` is called. All of the work of the application is performed by the subsidiary processes. These processes communicate with each other and with processes in other partitions using the APEX interface. C, C++ and Ada code can be mixed within an application, and there are various approaches available to manage the code. GNAT project files are the best method for combining mixed-language code within a partition.

Note that more complex organization of applications within a partition (e.g. multiple initialization threads) are possible, but require more complex setup and coordination. Refer to *VxWorks 653 Programmer's Guide*.

There are several examples of the basic application structure in the starter kit. Both C and Ada applications are provided.

E.4.2 The ARINC-653 APEX Binding

The Ada binding to the ARINC-653 APEX (APplication EXecutive) comes with the compiler and is available for all run-time libraries. The bindings are precompiled for each valid combination of run-time library and binding variant. Sources are found under '`<gnat-root>/include/apex/apex[_variant]/src/`'.

Static libraries that can be linked to are in '`<gnat-root>/lib/apex/<PLATFORM>-<RUNTIME>-<BI` For example, for VxWorks 653 for the PowerPC architecture, and the use of the cert run-time and Ada 83 version of the APEX binding, the location of the library to link is '`<gnat-root>/lib/apex/powerpc-wrs-vxworkae-cert-apex/libapex.a`'.

A GNAT project file to include in a project needing the APEX bindings is located at '`<gnat-root>/lib/gnat/apex.gpr`'. '`gprbuild`' and '`gnatmake`' know about this location, so to access this project file, add

```
with "apex";
```

to the top of your project file. This project file eliminates the need to explicitly include the desired APEX library into the `gnatlink` switches. Note that if you previously used '`environment.gpr`' in your project files for earlier versions of GNAT, it can still be used.

This project file takes three scenario variables:

- PLATFORM - the target triplet for your platform, e.g. powerpc-wrs-vxworksae. The default value is "powerpc-wrs-vxworksae".
- RUNTIME - the run-time library to use: zfp, cert, ravenscar-cert or full. The default value is "full".
- BINDING - the APEX binding to use: apex, apex_95, apex_zfp apex_zfp_95 or apex_minimal. The default value is "apex".

The bindings and valid combinations of RUNTIME and BINDING are described next.

Five bindings are provided:

- The standard Ada 83 APEX binding, designated by `BINDING=apex`. This binding is compatible with the full and cert run-time libraries. Applications using this binding must not use the Ada tasking facilities.
- The standard Ada 83 APEX binding, designated by `BINDING=apex_zfp`. This binding is compatible with the zfp, full and ravenscar-cert run-time libraries. In this case, applications using the full or ravenscar-cert libraries must not use any APEX processes.
- A subset of the standard APEX binding that corresponds to VTHREADS component APEX_MINIMAL, designated by `BINDING=apex_minimal`. This binding, rooted in unit `VxWorks_653.Apex_Minimal`, is suitable for use full and ravenscar-cert run-time libraries, when only APEX_MINIMAL is included in the application partition.
- The standard Ada 95/05 APEX binding, designated by `BINDING=apex_95`. This binding makes use of the hierarchical library facility of Ada 95 and Ada 2005. It also changes some definitions so that fewer explicit type conversions are needed in application code that uses the APEX binding. Wrapper packages are provided for backward compatibility with the Ada 83 binding. This binding is compatible with the cert and full run-time libraries. Applications must not use Ada tasking facilities.
- The standard Ada 95/05 APEX binding, designated by `BINDING=apex_zfp_95`. This binding is compatible with the zfp, full and ravenscar-cert run-time

libraries. In this case, applications using the full or ravenscar-cert libraries must not use any APEX processes.

The easiest way to set the binding is to add the project file '<gnat-root>/lib/gnat/apex.gpr' into the user project files via a context clause. GNAT will search and locate the path to this file if the environment variable GPR_PROJECT_PATH is defined to include '<gnat-root>/lib/gnat/' (this should be the case by default).

It is strongly recommended that you use the bindings provided with the product rather than alternatives, as the provided bindings have been extensively tested for correctness.

To access and use the apex project in an user project file:

```
with ''apex'';
project My_Project is
   ...
   for Source_Dirs use (<list of project source dirs>);
   -- To discriminate on eg variable Runtime:
   case apex.Runtime is
      when "cert" =>
      ...
   end case;
end My_Project;
```

E.4.2.1 Using the APEX Binding

The full bindings can be used if no Ada tasking constructs are used (note: ravenscar-cert applications should use apex_zfp, apex_zfp_95 or apex_minimal). The last is appropriate if only the APEX_MINIMAL component is included in the partition.

Alternatively, the zfp bindings can be used with the full and ravenscar-cert run-time libraries as long as no APEX processes are used.

E.4.2.2 Using Ada Processes

The standard way to define an APEX process in C is to create an attributes structure containing the process information (i.e. its name, entry point, priority, etc.). This record is then passed to the APEX CREATE_PROCESS routine.

It works the same way in Ada. One defines a record of type Apex_ Processes.Process_Attribute_Type describing the process. This record is then given as a parameter to Apex_Processes.Create_Process. Examples of how Ada processes are created are available in the starter kit.

The APEX / Ada binding provided with GNAT for VxWorks 653 adds the defaulted parameter Secondary_Stack_Size to the procedure Apex_ Processes.Create_Process. In order to support the use of unconstrained function results, a data structure called the secondary stack is allocated to each APEX process that executes Ada code. (The secondary stack is actually a mark/release heap). This parameter allows the user to specify the size of this data structure (in bytes). If the parameter is defaulted, the APEX binding will allocate a secondary stack with a size equal to one-fourth of the requested stack size for the process. This data structure is allocated out of the primary stack, which will have had its allocation increased by one-fourth, so that the original requested stack allocation is honored.

Calls to `Apex_Processes.Get_Process_Status` will return the size of the primary stack minus the size of the secondary stack (ie the size originally requested by the application developer).

Note that an APEX process created via the APEX Ada binding can query its secondary stack "high-water mark" using `GNAT.Secondary_Stack_Info.SS_Get_Max`. This package is described in *The GNAT Reference Manual*.

All APEX processes that execute Ada code, regardless of whether their bodies are largely in C or C++, must use the Ada binding routines to create the process or query its process status, or to create a process-level health monitoring handler. These routines and their parameters (for C) are described in '`apex_processes.ads`' and '`apex_health_monitoring.ads`'.

E.4.3 Selection of a Run-Time Profile

As mentioned GNAT High-Integrity Edition for VxWorks 653 provides several versions of the Ada Run-Time library suitable for differing certification and application needs.

These include:

- A full Ada run-time for application partitions, designated by the keyword "full".
- A restricted Ada run-time for application partitions, certified to DO-178B, Level A, designated by the keyword "cert".
- A minimal Ada run-time that generates no object code ("Zero FootPrint"), for use in either the Module OS or in application partitions, designated by the keyword "zfp".
- An implementation of the Ravenscar profile based on the cert profile for use in application partitions, designated by the keyword "ravenscar-cert".

The desired run-time library is selected at build time by setting the gnatmake flag '`--RTS=<run-time keyword>`', using one of the keywords given above. One can also import '`apex.gpr`', as described in the section on the APEX bindings, and define project variable `RUNTIME` using the '`-X`' switch.

The VxWorks 653 certified partition operating system supports the use of the restricted, Ravenscar and zero footprint profiles. For more information about profiles, see *The GNAT High-Integrity Edition Users Manual*.

E.4.4 Replacement of the Default Last Chance Handler

All Ada run-time libraries provided with the GNAT for VxWorks 653 include the concept of a last chance handler. This routine is called when an application terminates due to the occurrence of an unhandled Ada exception. All of the provided run-time libraries, except ZFP, provide a default implementation of the last chance handler. ZFP requires one to be written by the application developer.

The default handler can be overridden by the application developer in all cases, in order to provide such capabilities as raising an error to the health monitor. The following treatments apply:

- For the cert, ravenscar-cert and full run-time libraries, the default handler prints a stack dump and exception message, calls APEX RAISE_APPLICATION_ERROR, and then terminates the process in which the exception occurred.
- For the zfp run-time library, there is no default handler.

The profile of the last chance handler for all run-times except ZFP is:

```
procedure Ada.Exceptions.Last_Chance_Handler
  (Except :  Exception_Occurrence);
pragma Export (C,
               Last_Chance_Handler,
               "__gnat_last_chance_handler");
pragma No_Return (Last_Chance_Handler);
```

This handler may be replaced by any Ada or C routine that exports the symbol `__gnat_last_chance_handler` and that matches the given parameter profile.

The profile of the ZFP last chance handler is:

```
procedure Last_Chance_Handler
  (Source_Location : System.Address; Line : Integer);
pragma Export (C, Last_Chance_Handler,
               "__gnat_last_chance_handler");
```

The `Source_Location` parameter is a C null-terminated string representing the source location of the `raise` statement, as generated by the compiler, or a zero-length string if `pragma Discard_Names` is used. The `Line` parameter (when nonzero) represents the line number in the source. When `Line` is zero, the line number information is provided in `Source_Location` itself.

Again, any Ada or C routine that exports `__gnat_last_chance_handler` and matches the designated profile may be used here.

The recommended mechanism for replacing the last chance handler is to execute `powerpc-wrs-vxworksae-gnatmake` or the corresponding C compiler on the file providing `__gnat_last_chance_handler`. The resulting `.o` file should then be included in the linker directives for the Ada main application.

Note that the cert, full and ravenscar-cert run-times support stack overflow checking using the gcc switch `-fstack-check`. It is expected that a user-written last chance handler will be compiled without this switch (or that it will be overridden with `-fno-stack-check`). Additionally, user-written last chance handlers must not require more than 4KB of stack space so that they can handle a stack overflow Storage_Error.

E.4.5 Process-Level Health Monitoring Handler

The APEX routine `Apex_Health_Monitoring.Create_Handler` allows an application developer to provide a process-level handler for health monitoring events. The vThread that executes this routine will receive the requested stack size, plus 1/4 of the requested size for a secondary stack. This allows such routines to be written in Ada, though they need not be.

In addition to the standard HM events defined in APEX (`Apex_Health_Monitoring`), VxWorks 653 defines a number of extended HM codes. These are defined in '`$WIND_BASE/target/vThreads/h/hmTypes.h`'.

Appendix F VxWorks MILS Topics

This Appendix provides information specific to the GNAT cross-development system for the VxWorks MILS operating system.

F.1 Introduction and Overview

VxWorks MILS is a time- and space-partitioned real-time operating system supporting the development of security-sensitive systems requiring certification to high levels of assurance (EAL). It is based on a secure hypervisor / separation kernel that isolates and controls communication between a number of virtual boards. There are currently two kinds of virtual boards:

- Guest OS virtual boards running in a vThreads environment similar to that provided on VxWorks 653.
- The High Assurance Environment (HAE) for the stricter EALs, which provides a very restricted operating environment similar to a bare board.

It is anticipated that other OS's such as Wind River Linux will be provided as guest OS's as well.

GNAT for VxWorks MILS provides development tools for both kinds of virtual boards. Additionally, inclusion of compilation and analysis tools for the SPARK language supports the use of formal methods as required by higher EALs.

This version of GNAT targets the PowerPC PPC604 version of VxWorks MILS 2.1 or more recent (2.1.1 or later for e500v2 processors).

F.2 Common Considerations for VxWorks MILS Applications

These sections discuss system-wide subjects for VxWorks MILS.

F.2.1 Configuration

VxWorks MILS is highly configurable, covering such areas as scheduling, system call permissions for virtual boards, handling of exceptions and interrupts in virtual boards, communication between them, access to security and safety critical logs and memory allocation. The definitive reference to the configuration items for a VxWorks MILS system is *VxWorks MILS Configuration and Build Reference*. User guidance and explanations are provided in the *VxWorks MILS Configuration and Build Guide*. The Wind River documentation also includes a very useful tutorial *VxWorks MILS By Example*. The tutorial goes through configuration and build of an example system. Though the system is all written in C, it provides a very good overview of the process.

This document will not treat the overall configuration and build process, but will point out issues specific to mixed Ada, SPARK, C and C++ applications in the Guest OS and High Assurance environments.

F.2.2 GNATbench for MILS

GNATbench is the AdaCore plugin for Workbench. It provides a number of features supporting application development in Ada and SPARK. GNATbench recognizes Guest OS and HAE Workbench projects, and provides facilities to use these project types for Ada or mixed language development.

GNATbench uses the Workbench flexible build facilities for VxWorks MILS. For this reason, we generally suggest adding C or C++ sources directly to the virtual board project or one of its subfolders, and allowing the predefined build infrastructure to compile them. However, one can also have GNATbench manage their compilation using a GNAT project file. But in order to do so, the files should either be placed outside of the directories specified in `vpath` directives in the Makefile (including the project subtree), or have an extension distinct from the Workbench defaults of `.h`, `.c` and `.cc`. A naming convention is needed in the GNAT project file for the latter case.

An example mixed language system including both Guest OS and HAE virtual boards can be found in '`<GNAT root>\share\examples\powerpc-wrs-vxworksmils\mils_example`'. The example includes all necessary configuration files. A tutorial in the GNATbench Help menu in Workbench explains how to construct and build the system.

Using GNATbench to create virtual board projects that contain Ada code is the most straightforward method of integrating them into a MILS system.

F.2.3 Run-time Libraries for MILS

The following Ada run-time libraries are provided for MILS virtual boards:

- Zero Footprint ("zfp") provides minimal footprint run-time support. In its core configuration there is no contribution of object code to the application. Optional packages in the library provide useful facilities that will contribute object code. This run-time library can be used for SPARK applications. It is suitable for applications requiring certification to high EALs or to DO-178B level A. This is the only run-time library provided for the HAE.
- A restricted Ada library ("cert") for application partitions using APEX processes, certified to DO-178B Level A on VxWorks 653.
- Ravenscar Profile ("ravenscar-cert") implements the Ravenscar Ada tasking profile, along with a number of useful facilities for porting applications from other operating environments. It is based on the VxWorks 653 "cert" run-time library which has been certified to DO-178B Level A as part of aircraft

sub-systems. The profile supports up to EAL 4 and DO-178B Level A. It also supports the Ravenscar SPARK profile.

- Full Ada ("full") implements full Ada semantics and can be useful for porting existing applications into the Guest OS environment or for applications with relaxed or no certification requirements. It is not suitable for certification to DO-178B Level D or above.

For the Guest OS environment, the default run-time library is "full". For the HAE, the only alternative is "zfp".

For the Guest OS environment, different run-time libraries can be selected by withing '`apex.gpr`' in your GNAT project file and specifying the value of scenario variable `RUNTIME`. The possible values are as given above: one of "zfp", "cert", "ravenscar-cert" or "full". If not using a GNAT project file, the `--RTS` switch can be passed to `gnatmake` or one can specify the value for `gprbuild` to use with appropriate switches passed to `gprconfig`. See the *GNAT User's Guide, The GNAT Project Manager* for details. The allowable values for `--RTS` and `gprconfig` are the same as for scenario variable `RUNTIME`.

F.2.4 APEX Bindings

By default VxWorks MILS 2.1.1 provides a minimal subset of the ARINC-653 APEX facilities. It includes mainly queuing ports for both the Guest OS and the HAE. GNAT provides Ada 83 and Ada 95/05 style bindings to these facilities for both environments. These can be accessed by an application by withing '`apex.gpr`' in the GNAT project file, and specifying scenario variable `BINDING` to be `apex_mils` or `apex_mils_95`. The method for accessing these precompiled bindings without using a GNAT project file is described in this manual in the VxWorks 653 section *Using the APEX Binding*.

WRS has a patch available for the MILS 2.1.1 VxWorks Guest OS (CDR-R164191.1-1_101210_070131) that provides an almost full APEX Part 1 implementation (sampling ports and parts of HM are missing). GNAT includes full APEX bindings `apex` and `apex_95` that can be used with the cert or full run-time libraries if the patch is installed. These bindings cannot be used in applications that use Ada tasks.

F.2.5 Multiplexed I/O

VxWorks MILS provides a capability for multiplexing I/O among multiple virtual boards through a single serial channel. This facility is described in the *VxWorks MILS Configuration and Build Guide, AMIO Server Virtual Board* section. The monitor program is invoked on the host computer attached to the target serial port using '`%WIND_HOME%\vthreads-2.2.3\host\x86-win32\bin\wrmonitor.bat`'. It is also accessible in a Terminal window in Workbench.

A virtual board running the AMIO service must be part of the system. The sources and configuration files for such a partition can be found in '`<GNAT root>\share\examples\powerpc-wrs-vxworksmils\mils_example\amioServer`' and '`<GNAT root>\share\examples\powerpc-wrs-vxworksmils\mils_example\Integration\conf`
This partition has been extracted from the vThreads demo provided by WRS at '`%WIND_HOME%\vthreads-2.2.3\target\vThreads\demos\blasterDemoGuestOS`'.

Additionally, any Guest OS using AMIO must contain the files '`amioSio.c`' and '`usrAmioSerial.c`'. These appear in the '`receiver`' and '`sender`' directories of the example system, as well as in the WRS demo directory above.

An HAE application should include the files '`amioSio.c`', '`amioSio.h`' and '`usrAmio.c`' that can be found in the '`baremetal`' subdirectory of the GNAT MILS example.

Finally, the application main in a Guest OS VB should contain the following code. For a C main or in '`usrAppInit.c`', which is used to spawn an Ada main in a Guest OS virtual board:

```
void usrAppInit()

/******************************************************************************
 *
 * usrAppInit - initialize the user application
 */
#include <vxWorks.h>
#include <taskLib.h>

extern void sysSerialHwInit(void);
extern void sysSerialHwInit2(void);
extern void usrSerialInit(void);

/* ADA_MAIN is a macro defining the entry point of the Ada environment task
   in the virtual board Makefile
*/

extern void ADA_MAIN(void);

void usrAppInit()
{
/* MAIN_TASK_NAME is a macro defining the name to be displayed for the Ada
   environment task in the virtual board Makefile
*/
         int stackSize = 0x80000;
         int spawnFlags = VX_FP_TASK;
         char * mainTaskName = (char *) MAIN_TASK_NAME;
         int priority = 100;

         sysSerialHwInit();
         sysSerialHwInit2();
```

```
            usrSerialInit();



            /* Spawn Ada main */

            taskSpawn(mainTaskName, priority, spawnFlags, stackSize,
                            (FUNCPTR) ADA_MAIN, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
   }
```

For an Ada main in the HAE:

```
procedure My_Main is
   procedure usrAmioInit;
   pragma Import (C, usrAmioInit, "usrAmioInit");
...
begin
   usrAmioInit;
...
end;
```

F.2.6 Communication between Virtual Boards

VxWorks MILS provides ports and channels for communication between virtual boards. These are described in the *VxWorks MILS VxWorks Guest OS Guide*. Use of ports and channels requires configuration items in the '`VirtualBoard`' and '`Channels`' configuration documents respectively.

GNAT provides access to APEX queuing ports through the apex_mils and apex_mils_95 bindings. See the *VxWorks 653 Topics, Using the APEX Binding* section of this manual for instructions on accessing the APEX bindings.

In order to have consistent message passing behavior between virtual boards, the following fragment should appear in the MILS kernel configuration document.

```
        <Interrupts NumInterrupts="4">
                <!-- Global interrupt numbers must be unique -->

                <!-- Software interrupt mapping -->
                <Int Name="vbInterProcessorInt" GlobalIntNumber="3" />
                <Int Name="vbPortInt" GlobalIntNumber="4" />
                <Int Name="vbSharedMemoryInt" GlobalIntNumber="5" />
                <Int Name="vbScheduleStartInt" GlobalIntNumber="6" />
        </Interrupts>
```

For the same reason, the following should appear in the virtual board configuration documents.

```
        <Interrupts NumIn="1" NumOut="0">
                <In Name="vbScheduleStartInt" Vector="1" />
        </Interrupts>
```

F.2.7 Debugging on MILS

Debugging on VxWorks MILS requires the use of a dedicated virtual board. This is described in the *VxWorks MILS Configuration and Build Guide*. GNAT does not provide a port of gdb for this target, so debugging using the VxWorks MILS debug agent from Workbench is the standard method.

F.2.8 Binding to MILS Header Files

If one wishes to use APIs provided by the MILS header files directly, GNAT provides a binding generator switch for powerpc-wrs-vxworksmils-gcc and, if you have a native version of GNAT installed, for g++. The switch is `-fdump-ada-spec`, and is described in the *GNAT User Guide, Section 25: Generating Ada Bindings for C and C++ headers*.

F.3 Applications in the Guest OS Environment

F.3.1 Differences Between VxWorks 653 and VxWorks MILS vThreads Implementations

The VxWorks MILS guest OS is based on the VxWorks 653 vThreads Partition Operating System. However, it has a number of differences:

- Asynchronous virtual interrupts are provided in VxWorks MILS. These are enabled in the '`VirtualBoardDescription`' document. Therefore one can use the capabilities of `Ada.Interrupts` and the other interrupt handling capabilities described in the *Common VxWorks Topics* section of this manual.
- A Safety Critical log capability replaces Health Monitoring. Therefore the APEX health monitoring binding is not provided for MILS, and the default Ada last chance handler does not propagate unhandled exceptions to the health monitoring system. The user can override the default handler to log events to the safety critical log using `safeCritEventInject()` from '`safeCrit.h`'.
- VxWorks MILS provides only queuing ports and sampling ports, so these are the only APEX facilities provided in the apex_mils and apex_mils_95 Ada bindings.
- VxWorks MILS does not use stack guard pages. GNAT uses stack limit checking to check for stack overflow. This is still enabled with the compiler switch `-fstack-check`.
- VxWorks MILS does not provide hardware floating point overflow exceptions. Therefore, such overflows for unconstrained floating point types are

not passed to the Ada run-time system. If you want overflow detection, use a constrained floating point type.

- Hardware exceptions are not automatically passed to the guest OS. See the section on exception handling below for details.
- The root task in the guest OS (tRootTask) has a fixed 20KB stack and runs at the highest priority in the OS. Therefore, we use an application stub to spawn the Ada environment task as described in the *Application Stub* section below.
- By default, scheduling of vThreads in the VxWorks MILS Guest OS is equivalent to POSIX sched_fifo (Ada FIFO_Within_Priorities).

F.3.2 Ada Run-time Libraries for the Guest OS

Four Ada run-time libraries are provided for the guest OS: ZFP, Cert, Ravenscar and full Ada. These are described in the section above *Common Considerations for VxWorks MILS Applications*.

For the cert run-time library APEX processes written in C or C++ that call any Ada code must be created using the Ada binding in `apex` or `apex_95` and the Ada version of GET_PROCESS_STATUS must be used instead of the OS version. The prototypes for these routines are:

```
extern void CREATE_ADA_PROCESS (PROCESS_ATTRIBUTE_TYPE * ATTRIBUTES,
                     PROCESS_ID_TYPE * PROCESS_ID,
                     RETURN_CODE_TYPE * RETURN_CODE
                     STACK_SIZE_TYPE SECONDARY_STACK_SIZE);

extern void GET_ADA_PROCESS_STATUS (PROCESS_ID_TYPE PROCESS_ID,
                         PROCESS_STATUS_TYPE * PROCESS_STATUS,
                         RETURN_CODE_TYPE * RETURN_CODE);
```

`SECONDARY_STACK_SIZE` is the number of bytes to add to the requested stack size of the APEX process to provide a buffer used to return unconstrained Ada function results. If set to 0, 1/5 of the requested process stack size will be added to the process stack for this buffer. The buffer is checked and cannot overflow. If an allocation would cause it to overflow, an Ada `Storage_Error` exception is raised instead.

The relevant APEX library to link with can be found in: '`<GNAT root>\lib\apex[_95]\<processor>-wrs-vxworksmils-<runtime>-<binding>\libapex.a`'

For the full run-time library, tasks written in C or C++ can use the facilities of package `GNAT.Threads` to register with the Ada run-time library if they call Ada code.

For the Ravenscar run-time library, which requires static creation of tasks, tasks calling Ada and coded in C or C++ must be created on the Ada side, and then the Ada task can call the C or C++ entry point routine. Note that this approach can also be used with the full run-time library.

F.3.3 Exceptions and Exception Handling on the Guest OS

There are two considerations mentioned above that make exception handling different from the VxWorks 653 partition operation system: replacement of the health monitoring system by the safety critical log, and not passing hardware exceptions by default to the guest OS.

In the first case, the default last chance handler does not interact with the safety critical log as it did with the health monitoring system on VxWork 653. A solution is to override the default handler and log unhandled exceptions to the safety critical log using `safeCritEventInject()` from '`safeCrit.h`'. The compiler binding generator switch `-fdump-ada-spec` can be used to generate an Ada specification for this library. See *VxWorks 653 Topic, Application Design Considerations, Replacement of the Default Last Chance Handler* for the mechanics of replacing the handler.

For the second case, if one wants hardware exceptions to be mapped to Ada exceptions in the usual way, one must add a `PassExceptions` section to the '`VirtualBoard`' configuration document. The following XML fragment provides the usual mappings:

```
<PassExceptions NumExcs="4">
        <Exception Vector="0x300" />
        <Exception Vector="0x400" />
        <Exception Vector="0x700" />
        <Exception Vector="0x800" />
</PassExceptions>
```

F.3.4 Application Stub

Since tRootTask for the guest OS has a fixed 20KB stack and executes at the highest priority in the guest OS, the application developer should provide routine `usrAppInit()` in order to perform initializations such as for multiplexed I/O (AMIO) and to spawn the Ada environment task with a suitably sized stack. Note that enabling of stack overflow checking via compiler switch `-fstack-check` requires 12KB above the anticipated stack requirement of an application task in order to effectively handle stack overflows (not to be used with the ZFP run-time library). Additionally note that for the full and Ravenscar run-time libraries, the VxWorks priority given to taskSpawn for the environment task will be replaced during elaboration of the application. If no `pragma Priority` is given in the main subprogram, the default Ada priority of 122 (equivalent to VxWorks priority 133) will be used.

Here is a sample usrAppInit.c suitable for starting an Ada application:

```
#include <vxWorks.h>
#include <taskLib.h>
```

```
/* AMIO interface if present in the virtual board */

extern void sysSerialHwInit(void);
extern void sysSerialHwInit2(void);
extern void usrSerialInit(void);

/* Ada binder-generated main - ADA_MAIN is passed as a macro from the makefile */

extern void ADA_MAIN(void);

void usrAppInit()

{

        int stackSize = 0x80000;
        int spawnFlags = VX_FP_TASK;

         /* MAIN_TASK_NAME is passed as a macro from the makefile */
        char * mainTaskName = (char *) MAIN_TASK_NAME;
        int priority = 100;

         /* Initialize AMIO if interface is present in the virtual board */

        sysSerialHwInit();
        sysSerialHwInit2();
        usrSerialInit();


        /* Spawn Ada environment task */

        taskSpawn(mainTaskName, priority, spawnFlags, stackSize,
                        (FUNCPTR) ADA_MAIN, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
}
```

`ADA_MAIN` is the name of the Ada main subprogram in lower case (i.e. the symbol for the entry point of the Ada application). `MAIN_TASK_NAME` is the name you would like associated with the Ada environment task for the application. These are typically passed using a `-D<symbol>=<value>` switch to the compiler in the virtual board makefile.

The stack size can be varied to suit the needs of the environment task. It must be spawned with at least option `VX_FP_TASK`.

A copy of this file can be found in '`<GNAT root>\share\examples\powerpc-wrs-vxworksmils\mil`

F.3.5 Guest OS Virtual Board and Application Configuration Files

In addition to the `PassException` fragment in the '`VirtualBoard`' document, the following fragment should also be added to ensure consistent message passing behavior. Other interrupts can be added to the section.

```
<Interrupts NumIn="1" NumOut="0">
        <In Name="vbScheduleStartInt" Vector="1" />
</Interrupts>
```

Otherwise, nothing is different in the configuration files from those for C or C++ applications. '`VirtualBoard`', '`GuestOS`' and application '`Payload`' documents are written, and entries are made in various system level configuration documents.

Refer to the *VxWorks MILS Configuration and Build Guide* for instructions on configuring virtual boards and integrating them into a MILS system.

Note that a full example with configuration files is provided in '`<GNAT root>\share\examples\powerpc-wrs-vxworksmils\mils_example`'.

F.3.6 Setting Up a Guest OS Application with GNATbench

GNATbench, the AdaCore plugin for Workbench, supports building applications for the VxWorks virtual board environment and for the HAE. An example system is provided in directory '`<GNAT root>\share\examples\powerpc-wrs-vxworksmils\mils_example`'. Instructions are given in the GNATbench documentation that can be found under the Workbench Help menu.

In general, for Guest OS applications, one creates a Guest OS Application Project in Workbench, then imports sources, application configuration files, and at least a root GNAT project file. Then one converts the Workbench project for use with GNATbench. For Vxworks MILS, the Workbench "flexible managed build" scheme is used. When the project is converted, GNATbench generates '`gpr-mils.makefile`' and '`scenario.makefile`' into the project root directory. The first contains rules and definition relevant to using gprbuild to build the project application code. The second contains application-specific definitions, including the values of scenario variables defined in the GNAT project file. This file is updated whenever the user modifies values in the GNATbench scenario view. The two files are then used with the Workbench generated Makefile for the project at build time.

Note that the name of the root GNAT project file need not be the same as the name of the Workbench project.

F.3.7 Guest OS Application Makefiles

If building a Guest OS virtual board directly using Makefiles, the following files can be adapted for the application build part.

This first file is like the 'scenario.makefile' generated by GNATbench:

```
# user variables

# Path to root GNAT project file. Use an absolute path.
GPRPATH=C:/Workbench-mils-2.0.1-rc2/workspace/receiver/receiver.gpr

# Name of application / virtual board project
APP_NAME=receiver

# Main subprogram name in lower case
MAIN=tf_receiver_init

# Path containing project. Use an absolute path.
PRJ_ROOT_DIR=

# APEX binding to use. One of: apex_mils, apex_mils_95
BINDING=apex_mils

# Ada run-time library to use.  One of: zfp, full, ravenscar-cert
RUNTIME=full

# Not to be modified by user:
PLATFORM=powerpc-wrs-vxworksmils
GNAT_SCENARIO_COMMAND=-XBINDING=$(BINDING) -XRUNTIME=$(RUNTIME) -XPLATFORM=$(PLATF
```

The second file contains rules for building with gprbuild. 'gpr-mils.makefile' can be copied from an existing Guest OS application project that has been converted for GNATbench use, as no modifications should be needed other than adaptation to specifics of your build infrastructure.

```
-include $(PRJ_ROOT_DIR)/scenario.makefile
# GPR boilerplate to be added to vthreads Guest OS projects.
#
# Depends on the following symbols being defined.
#
# Extracted by GNATbench from the .gpr fragment or when extending the project:
#   MAIN - entry point for the application main
#   GPRPATH  - location of the root GNAT project fragment
#
# Scenario variables:
#   PLATFORM - tool prefix triplet, eg powerpc-wrs-vxworksae. GNATbench should be
#               determine this without an explicit scenario variable.
#   GNAT_SCENARIO_COMMAND - values of scenario variables found in GPR fragment
#
# Macros:
#    APP_NAME - name of the project
#

# GPR related definitions:

# Because usrRootTask stack is fixed at 20K, modify usrAppInit to spawn the Ada ma
# the specified stack size
ADDED_CFLAGS = -DADA_MAIN=$(MAIN) -DMAIN_TASK_NAME="$(MAIN)"
LDFLAGS_EXTRA=--section-start .text=0x10000 -e 0x10000

# Object directory relative to GPR fragment location. Needed because the gprbuild
# requires a location expressed relative to the GPR fragment.
PM_OBJ_SUBDIR = $(CPU)gnu/$(APP_NAME)-pm/$(MODE_DIR)/Objects

# Absolute path of the project's object directory
PM_OBJ_DIR = $(PRJ_ROOT_DIR)/$(PM_OBJ_SUBDIR)

# Absolute path to GPR config fragment
CONFIG_FILE = $(PM_OBJ_DIR)/$(PLATFORM).cgpr

# Add main generated by gprbuild and project objects list
MAIN_OBJECT = $(PM_OBJ_DIR)/$(MAIN).o
OBJECTS_$(APP_NAME)-pm += $(MAIN_OBJECT)

# Configure Ada, C and C++ tools for gprbuild
$(CONFIG_FILE) : FORCE
        if [ ! -d "`dirname "$@"`" ]; then mkdir -p "`dirname "$@"`"; fi;
        gprconfig --target=$(PLATFORM) --config=ada,,$(RUNTIME), \
          --config=c,,,,"GCC-WRSMILS" --config=c++,,,,"G++-WRSMILS" --batch \
                 -o $(CONFIG_FILE)

# Build step:
.PHONY: FORCE
FORCE:

external_build ::

$(PROJECT_TARGETS): $(MAIN_OBJECT)
```

```
$(MAIN_OBJECT) : $(CONFIG_FILE) FORCE
        @echo "make: building GPR objects"
        if [ ! -d "`dirname "$@"`" ]; then mkdir -p "`dirname "$@"`"; fi;
        gprbuild -p -P "$(GPRPATH)" \
           -o $(PM_OBJ_SUBDIR)/$(MAIN).o \
           --config=$(CONFIG_FILE) \
           $(GNAT_SCENARIO_COMMAND)

# Clean step:
external_clean::
        @echo "make: cleaning GPR objects"
        if [ `ls $(CONFIG_FILE)` ]; then \
        gprclean -r -P "$(GPRPATH)" \
          --config=$(CONFIG_FILE) \
           $(GNAT_SCENARIO_COMMAND); \
           $(RM) $(CONFIG_FILE); \
           fi;
```

F.4 Applications in the High Assurance Environment

GNAT can be used to build Ada, SPARK and mixed-language applications for execution in the High Assurance Environment (HAE). SPARK Pro can be used to apply formal methods as required by EALs 5 and higher to SPARK code.

F.4.1 Ada Run-time Libraries for the HAE

Only the Zero FootPrint run-time library (ZFP) is appropriate for Ada or SPARK applications that will execute in the HAE. See the *GNAT User's Guide Supplement for High-Integrity Edition Platforms, Section 3.2, The Zero Footprint Profile* for a description of the Ada language features supported by this run-time library.

F.4.2 Exceptions and Exception Handling in the HAE

The ZFP run-time library provides two facilities for handling Ada exceptions: handling an exception within a local scope, and the use of a user-written last chance handler for unhandled exceptions.

The first facility allows the writing of exception handlers for a given scope. Any propagation beyond the scope within which an exception is raised goes to the last chance handler.

The last chance handler has the following profile:

```
with System;

package Last_Chance_Handler is
```

```
     procedure Last_Chance_Handler (Msg : System.Address; Line : Integer);
     pragma Export (C, Last_Chance_Handler, "__gnat_last_chance_handler");
     pragma No_Return (Last_Chance_Handler);

  end Last_Chance_Handler;
```

It may be used to interact with the security and safety critical logs or to take other critical actions such as partition restart.

Hardware exceptions can be handled in the HAE using `excConnect` from '`excLib.h`' to associate a procedure for handling an exception. The `-fdump-ada-spec` switch to '`powerpc-wrs-vxworksmils-gcc`' (or to '`g++`' if a native version of GNAT is intalled) can be used to generate an Ada binding to the library headers. As for Guest OS virtual boards, a `PassExceptions` section must be specified in the '`VirtualBoard`' document for the HAE. See the *VxWorks MILS High Assurance Environment API Reference* for a description of '`excLib`'.

Hardware exceptions are not mapped to Ada exceptions in the HAE.

F.4.3 Security and Safety Critical Logs

The HAE is a common place to monitor the security and safety critical logs. The Blaster demo '`main.c`' provided by WRS illustrates this for the security log. The relevant libraries are '`safeCrit.h`' and '`secAudit.h`'. Their use is described in the *VxWorks MILS High Assurance Environment API Reference*.

F.4.4 HAE Virtual Board Configuration File

An HAE virtual board uses a '`VirtualBoard`' configuration document, but unlike the Guest OS, does not use a '`GuestOS`' document. There are no special configuration considerations for Ada applications in the HAE.

F.4.5 Setting Up an HAE Application with GNATbench

GNATbench can be used to set up an HAE virtual board project similarly to the way it is done for a Guest OS virtual board. The example mentioned previously demonstrates how to do this in conjunction with the GNATbench documentation available under the Workbench Help menu.

For the HAE, one starts with a Workbench High Assurance Environment Application project, then imports sources and a root GNAT project file, then converts the project for GNATbench use. Files '`scenario.makefile`' and '`hae-mils.makefile`' are generated as part of the conversion, and '`scenario.makefile`' is kept in synch with the GNATbench scenario view.

No application stub is needed for the HAE, but note that the entry point to the Ada application is always `main`. If you are using AMIO in the HAE, it should be initialized in the Ada main subprogram analogously to the way it's done in 'usrAppInit.c' for the Guest OS. GNAT can generate an Ada binding as mentioned previously.

F.4.6 HAE Application Makefiles

The makefiles for the HAE are similar to those for the Guest OS. Some minor differences occur in the gprbuild rules.

'scenario.makefile':

```
# user variables

# Path to root GNAT project file. Use an absolute path.
GPRPATH=C:/Windriver/workspace/baremetal/baremetal.gpr

# Name of application / virtual board project
APP_NAME=baremetal

# Main subprogram name
MAIN=baremetal

# Path containing project. Use an absolute path.
PRJ_ROOT_DIR=
_
# APEX binding to use. One of: apex_mils, apex_mils_95
BINDING=apex_mils

# Not to be modified by user:

# Ada run-time library to use.
RUNTIME=zfp

PLATFORM=powerpc-wrs-vxworksmils
GNAT_SCENARIO_COMMAND=-XBINDING=$(BINDING) -XRUNTIME=$(RUNTIME) -XPLATFORM=$(PLATF
```

The second file contains rules for building with gprbuild. 'hae-mils.makefile' can be copied from an existing HAE application project that has been converted for GNATbench use, as no modifications should be needed other than adaptation to specifics of your build infrastructure.

'hae-mils.makefile':

```
-include $(PRJ_ROOT_DIR)/scenario.makefile
# GPR boilerplate to be added to application makefiles when .wrmakefile is
# processed for MILS HAE environment projects. GNATbench generates this fragment
# into the top-level of the project.
#
# Depends on the following symbols being defined.
```

```
#
# Extracted by GNATbench from the .gpr fragment or when extending the project:
#   MAIN - name of the application main, typically a stub written in C that calls
#          an Ada main.
#   GPRPATH  - location of the root GNAT project fragment
#
# Scenario variables:
#   PLATFORM - tool prefix triplet, eg powerpc-wrs-vxworksae. GNATbench should be
#              determine this without an explicit scenario variable.
#   GNAT_SCENARIO_COMMAND - values of scenario variables found in GPR fragment
#
# Macros (for now):
#    APP_NAME - name of the project
#

# GPR related definitions:

# Object directory relative to GPR fragment location. Needed because the gprbuild
# requires a location expressed relative to the GPR fragment.
OBJ_SUBDIR = $(APP_NAME)/$(MODE_DIR)/Objects/$(APP_NAME)

# Path to GPR config fragment
CONFIG_FILE = $(OBJ_SUBDIR)/$(PLATFORM).cgpr

# Add main generated by gprbuild to project objects list
MAIN_OBJECT = $(PRJ_ROOT_DIR)/$(BUILD_SPEC)/$(OBJ_SUBDIR)/$(MAIN).o
OBJECTS_$(APP_NAME) += $(MAIN_OBJECT)

# Configure Ada and C tools for gprbuild
$(CONFIG_FILE) : FORCE
        if [ ! -d "`dirname "$@"`" ]; then mkdir -p "`dirname "$@"`"; fi;
        gprconfig --target=$(PLATFORM) --config=ada,,$(RUNTIME), \
          --config=c,,, --batch \
                 -o $(CONFIG_FILE)

# Build step:
.PHONY: FORCE
FORCE:

external_build ::

$(PROJECT_TARGETS): $(MAIN_OBJECT)
```

```
$(MAIN_OBJECT) : $(CONFIG_FILE) FORCE
        @echo "make: building GPR objects"
        if [ ! -d "`dirname "$@"`" ]; then mkdir -p "`dirname "$@"`"; fi;
        gprbuild -p -P "$(GPRPATH)" \
           -o $(PRJ_ROOT_DIR)/$(BUILD_SPEC)/$(OBJ_SUBDIR)/$(MAIN).o \
           --config=$(CONFIG_FILE) \
           $(GNAT_SCENARIO_COMMAND) \
           -bargs -Mmain

# Clean step:
external_clean::
        @echo "make: cleaning GPR objects"
        if [ `ls $(CONFIG_FILE)` ]; then \
        gprclean -r -P "$(GPRPATH)" \
          --config=$(CONFIG_FILE) \
           $(GNAT_SCENARIO_COMMAND); \
           $(RM) $(CONFIG_FILE); \
           fi;
```

Appendix G LynxOS Topics

This chapter describes topics that are specific to the GNAT for LynxOS cross configurations.

G.1 Getting Started with GNAT on LynxOS

This section is a starting point for using GNAT to develop and execute Ada programs for LynuxWorks' LynxOS target environment from a Unix host environment. We assume that you know how to use GNAT in a native environment and how to start a telnet or other login session to connect to your LynxOS board.

To compile code for a LynxOS system running on a PowerPC board, the basic compiler command is `powerpc-elf-lynxos-gcc`.

With GNAT, the easiest way to build the basic `Hello World` program is with `gnatmake`. For the LynxOS PowerPC target this would look like:

```
$ powerpc-elf-lynxos-gnatmake hello
powerpc-elf-lynxos-gcc -c hello.adb
powerpc-elf-lynxos-gnatbind -x hello.ali
powerpc-elf-lynxos-gnatlink hello.ali
```

(The first line is the command entered by the user – the subsequent three are the programs run by `gnatmake`.)

This creates the executable `hello`, which you then need to load on the board (using ftp or an NFS directory for example) to run it.

G.2 Kernel Configuration for LynxOS

The appropriate configuration for your LynxOS kernel depends on the target system and the requirements of your application. GNAT itself adds no additional demands; however in some situations it may be appropriate to increase the conservative resource assumptions made by the default configuration.

Kernel parameters limiting the maximum number of file descriptors, kernel and user threads, synchronization objects, etc., may be set in the file '`uparam.h`'. You may also wish to modify the file '`/etc/starttab`', which places limits on data, stack, and core file size. See the documentation provided by LynuxWorks for more information.

G.3 Debugging Issues for LynxOS

GNAT's debugger is based on the same GNU gdb technology as the debugger provided by LynxOS, though with a great number of extensions and enhancements to support the Ada language and GNAT. The LynxOS documentation is relevant to understanding how to get the debugger started if you run into difficulties.

G.3.1 Native Debugging on x86 LynxOS

Although there is no native GPS port to LynxOS it is possible to use GPS from a remote host, for instance from Solaris.

You will need to have your working directory mounted on both the host and the target, and you will need to have a remote shell utility like `rsh` set up to allow password-free execution of commands with the correct environment on the target. You will also need a project file for the program you wish to debug.

You can then:

1. Open your project on the host from GPS.
2. Open the project editor: `Project -> Edit_Project_Properties`
3. For the "General" tab, enter the name of the LynxOS native machine in the "Tools host" text box
4. Exit the project editor
5. Open the `Preferences` editor: `Edit -> Preferences`
6. `>`From the Debugger page, disable the option "Execution Window"

You should then be able to build, execute, and debug your program from GPS.

G.3.2 Cross Debugging on PPC LynxOS

The procedure for cross debugging on LynxOS is similar, but requires two additional steps. First, the executable must be launched on the target under the utility `gdbserver`, and then the cross debugger must be started on the host and attached to it.

To demonstrate a debugging session, we will use a slightly more complex program called '`demo1.adb`', which can be found in the '`examples`' directory of the GNAT distribution. This program is compiled with debugging information as follows:

```
$ powerpc-elf-lynxos-gnatmake -g demo1
powerpc-elf-lynxos-gcc -c -g demo1.adb
powerpc-elf-lynxos-gcc -c -g gen_list.adb
powerpc-elf-lynxos-gcc -c -g instr.adb
powerpc-elf-lynxos-gnatbind -x demo1.ali
powerpc-elf-lynxos-gnatlink -g demo1.ali
```

Once the executable is created, copy it to your working directory on the board. In this directory, you will have to launch the gdb server and choose a free port number on your TCP/IP socket. Presuming the Internet hostname of the board is '`myboard`' and the port chosen is 2345, issue the following command:

```
myboard> gdbserver myboard:2345 demo1
```

Then return to your host environment.

Next, attach from gdb:

```
(gdb) file my_program
(gdb) target remote myboard:2345
```

To run the cross debugger from the command line without the visual interface use the command `powerpc-elf-lynxos-gdb`.

You will see something like:

```
GNU gdb 4.17.gnat.3.14a1
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.  There is absolutely no warranty
for GDB.  Type "show warranty" for details.  This GDB was configured as
"--host=sparc-sun-solaris2.5.1 --target=powerpc-elf-lynxos".
(gdb)
```

where `(gdb)` is the debugger's prompt. The first thing to do at the prompt from within `gdb` is to load the symbol table from the executable:

```
(gdb) file demo1
Reading symbols from demo1...done.
(gdb)
```

You then have to attach to the server running on the board. Issue the command:

```
(gdb) target remote myboard:2345
```

After the server has been started and attached from the host, the program is running on the target but has halted execution at the very beginning. The following commands set a breakpoint and continue execution:

```
(gdb) break demo1.adb:37
Breakpoint 1 at 0x100064d0: file demo1.adb, line 37.
(gdb) cont
Continuing.

Breakpoint 1, demo1 () at demo1.adb:37
37         Set_Name (Fuel, "Fuel");
(gdb)
```

Here the execution has stopped at the breakpoint set above. Now you can use the standard `gdb` commands to examine the stack and program variables.

Note that once execution has completed, the server on the board must be restarted before a new debugging session may begin.

G.4 An Example Cross Debugging Session for LynxOS

Carrying on a little further with the debugging session, the following example illustrates some of the usual debugging commands for moving around and seeing where you are:

```
(gdb) next
38         Set_Name (Water, "Water");
(gdb) bt
```

```
#0  demo1 () at demo1.adb:38
#1  0x10001218 in main (argc=1, argv=2147483640, envp=2147483520) at
b~demo1.adb:118
#2  0x10017538 in runmainthread ()
#3  0x10001048 in __start ()
(gdb) up
#1  0x10001218 in main (argc=1, argv=2147483640, envp=2147483520) at
b~demo1.adb:118
118       Ada_Main_Program;
(gdb) down
#0  demo1 () at demo1.adb:38
38         Set_Name (Water, "Water");
(gdb)
```

To examine and modify variables (of a tagged type here):

```
(gdb) print speed
$1 = (name => "Speed         ", value => -286331154)
(gdb) ptype speed
type = new instr.instrument with record
    value: instr.speed;
end record
(gdb) speed.value := 3
$2 = 3
(gdb) print speed
$3 = (name => "Speed         ", value => 3)
(gdb) info local
speed = (name => "Speed         ", value => 3)
fuel = (name => "Fuel          ", value => -286331154)
oil = (name => ' ' <repeats 14 times>, value => -286331154, size => 20,
  fill => 42 '*', empty => 46 '.')
water = (name => ' ' <repeats 14 times>, value => -286331154, size => 20,
  fill => 42 '*', empty => 46 '.')
time = (name => ' ' <repeats 14 times>, seconds => 0, minutes => 0, hours =>
0)
chrono = (name => ' ' <repeats 14 times>, seconds => 0, minutes => 0,
  hours => 0)
db = (access demo1.dash_board.internal) 0x0
(gdb)
```

And finally letting the program it run to completion:

```
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

G.5 Issues Linking on LynxOS

On PPC Lynx, certain system libraries contain 24 bit PC relative jump instructions. For very large applications it may be impossible for the linker resolve some relocations at link time.

In this case you will see an error like the following:

```
<object file> : relocation truncated to fit: R_PPC_REL24 exit
collect2: ld returned 1 exit status
```

One solution to this problem is to reduce the size of your binary. There are several approaches you can try, for instance:

- Turn off run time checks. (-gnatp)
- Optimize for space rather than time. (-Os)
- Disable stack checking (remove option -fstack-check)
- Disable inlining. (-fno-inline)

If these workarounds are not sufficient to link your application, please contact AdaCore support at report@adacore.com for additional help.

Appendix H AAMP Topics

This chapter describes topics that are specific to the GNAT-for-AAMP cross configurations.

H.1 Getting Started with GNAT for AAMP (GNAAMP)

This section is a starting point for using GNAT for AAMP (GNAAMP) to develop and execute Ada programs for the Rockwell Collins AAMP target environment from a Windows host environment. We assume that you know how to use GNAT in a native environment and how to download a linked AAMP executable to the AAMP target hardware or run it under control of Facade.

First it's important to understand the naming conventions used for the GNAAMP compiler and tools, which differ from the conventions normally used for GNAT cross compilers. The GNAAMP compiler itself has the name `gnaamp` rather than `gcc`, as this compiler does not use the gcc driver and back end, but rather has its own customized back end. The other tools that come with the GNAAMP cross-compiler are generally named by substituting `gnaamp` for the usual `gnat` tool prefix. For example, the name of the program build tool is `gnaampmake` rather than `gnatmake`. Similarly, the GNAAMP binder is named `gnaampbind` and the GNAAMP linker is `gnaamplink`.

To compile code for an AAMP target, the basic compiler command is `gnaamp`, but the easiest way to build the basic `Hello World` program is by running `gnaampmake`, which results in output similar to the following:

```
$ gnaampmake hello
gnaamp -c hello.adb
assembling hello.asm ...
MACASM  CPN 613-3583-006  8-Oct-2002
Program Size in Words for Counter 1, Code     = 0009 Hex, 9 Decimal.
Program Size in Words for Counter 2, LitData = 0006 Hex, 6 Decimal.
End MACASM.  0 Errors, 0 Warnings, 0 Messages.
gnaampbind -x hello.ali
gnaamplink hello.ali
assembling b~hello.asm ...
MACASM  CPN 613-3583-006  8-Oct-2002
Program Size in Words for Counter 0, VolData = 0001 Hex, 1 Decimal.
Program Size in Words for Counter 1, Code     = 0068 Hex, 104 Decimal.
Program Size in Words for Counter 2, LitData = 0030 Hex, 48 Decimal.
End MACASM.  0 Errors, 0 Warnings, 0 Messages.
clink -noupper -full hello.lec
CAPS Link Editor 2.0  03-Jan-2002  CPN xxx-xxxx-xxx
End Link.  0 Errors, 0 Warnings, 0 Messages.
```

The first line is the command entered by the user, and the subsequent lines are produced by the programs run by `gnatmake`. This creates the executable `hello.axe`. Running `gnatmake` will not produce an ftt file, so if that file is needed by your AAMP execution environment then you must create it yourself (e.g., by creating a file named hello.ftt that contains a single at-sign character).

Note that the AAMP macro assembler (`macasm`) and the AAMP linker (`clink`) are also provided as part of the cross compiler installation, and are invoked by `gnaamp` and `gnaamplink` respectively. The AAMP assembler is called from the compiler with the following command:

```
macasm -in_mac <path>\lib\gnaamp\aamp5 -noupper -instr -cross hello.asm
```

where <*path*> indicates the GNAAMP installation directory.

The AAMP linker is invoked from `gnaamplink` via an intermediate tool named `aalink` using the following command:

```
clink -noupper -full hello.lec
```

Another tool worth knowing about is gnaampcmd, which is the GNAAMP-specific version of the GNAT command driver. This GNAAMP command driver can be used to invoke other tools by giving it a tool designator as an argument (for example, MAKE, LINK, PP, etc.). It also supports the use of project files in conjunction with tools that don't support them directly, such as gnaamppp (the GNAAMP version of the gnatpp pretty printer).

As an example, the following project file specifies various switches to pass to certain of the tools, such as gnaampmake and the gnaamp compiler, as well as defining the mappings of various tools to their corresponding GNAAMP names. The project file would reside in a file named gnaamp_app.gpr, with sources located, in this case, in the same directory as the project file.

```
project GNAAMP_App is

   for Source_Dirs use (".");
   for Object_Dir use "obj";
   for Main use ("main_program.adb");

   package Builder is
      for Default_Switches ("ada") use ("-g", "-s",
                                    "-aamp_target=aamp5_small");
   end Builder;

   package Compiler is
      for Default_Switches ("ada") use ("-g", "-O1", "-univ");
   end Compiler;

   package IDE is
      for Compiler_Command ("ada") use "gnaampmake";
      for Gnatlist use "gnaampls";
      for Gnat use "gnaampcmd";  -- needed to invoke gnaamppp
   end IDE;

end GNAAMP_App;
```

This project file can then be specified when invoking gnaampmake to build the application main_program:

```
gnaampmake -Pgnaamp_proj
```

You can use the following command to run gnaamppp with this project:

```
gnaampcmd -Pgnaamp_proj PP
```

For more information on how to use GNAT tools and project files, see *The GNAT User's Guide*.

H.2 GNAAMP-Specific Switches

The switches documented for the GNAT compiler and related tools are generally available for GNAAMP, but note that switches that are specific to the GCC back end are typically not applicable to the GNAAMP compiler. The debugging switch `-g` and the optimization switch `-O` are supported, but currently only one level of optimization is available, so it suffices to apply the `-O1` switch to enable optimization for the `gnaamp` and `gnaampmake` commands (i.e., `-O2` and `-O3` have no additional effect beyond `-O1`).

The following AAMP-specific switches are supported:

`-aamp_target=`*`target_name`*
: This switch allows users to specify an alternative AAMP target library. The name given as the switch argument is used to determine the name of the target library subdirectory under the GNAAMP installation's `lib` directory, as well as indicating the name to use for the target's macro library. For example, specifying `-aamp_target=aamp7` on the `gnaampmake` command directs the compilation tools to locate library sources and objects under the subdirectory `\lib\aamp7` and to use the macro library named `aamp7.mlb` in that same subdirectory. An alternative means of specifying the target library is to set the environment variable `aamp_target` to the desired target name. The use of the `-aamp_target` switch takes precedence over the environment variable. In the absence of a user specification by the switch or variable, the default for the target name is `aamp5`. In addition to giving the switch on the `gnaamp` and `gnaampmake` commands, `-aamp_target` is also allowed on invocations of `gnaamplink`. The only way to specify the target name on a direct invocation of the `gnaampbind` command is to set the `aamp_target` environment variable.

`-macasm="`*`arguments`*`"`
: This switch supports passing customized arguments through to the AAMP macro assembler. All arguments to macasm are enclosed in double quotes. For example, to pass the `-timer` argument to

macasm: `-macasm="-timer"`. Note that the macasm switch `-in_mac` can be passed explicitly using `-macasm` to override the default value specified for that switch by the compiler.

`-O`
: This switch enables the peephole optimization phase in the GNAAMP back end. The compiler supports passing the optimization switch with an indicated level (-O1, -O2, or -O3), but all three of these levels are currently equivalent to passing -O.

`-univ`
: This switch enables generation of universal instructions for addressing global data objects. Application of the switch on the compilation command is equivalent to giving the pragma `Universal_Data` on any compiled units (see the *GNAT Reference Manual*).

H.3 Clock Support

The GNAAMP implementation does not provide direct support for a real-time clock. Instead, there is a default dummy version of the function _APP_GET_TIME, exported by the AAMP run-time library, that is used for programs that make use of Ada.Calendar. The dummy implementation is not a real clock function and simply increments an internal Duration variable by 1/100th of a second each time it is called. There is also a procedure exported as _APP_SET_TIME that allows setting the initial time value. Both of these routines are declared as part of the Mini_RTE package that is linked with all programs (see the `"rts"` subdirectory in your installation directory). The body of the subunit Mini_RTE.Clock must be customized for an application to provide a different implementation of these routines.

H.4 Debugging for GNAAMP

It is assumed that the user is familiar with the Facade debugging environment available on Windows for debugging AAMP programs. When compiling Ada programs to be debugged on GNAAMP, we recommend using the `-g` switch, which enables the generation of debugging information for each unit compiled with the switch. The `-g` switch causes the GNAAMP compiler to include source line information in object files and to generate a Debugger Symbol Table file ('`.dst`' file) containing high-level symbol information for the compiled unit. See the *Facade User's Guide* published by Rockwell Collins, Inc., for further information on debugging.

Appendix I PowerPC 55xx ELF Topics

This Appendix describes topics that are specific to the GNAT for PowerPC 55xx ELF cross configurations.

I.1 Introduction

The PowerPC 55xx ELF toolset targets Book E variants of the PowerPC architecture. It was originally developed for the Freescale e200 core used in the 5554 microcontroller.

The prefix generally used for target-specific executables in the PowerPC 55xx ELF toolset is *powerpc-eabispe*. E.g., the compiler is `powerpc-eabispe-gcc` and the binder is `powerpc-eabispe-gnatbind`. The one exception is the debugger, which is shared with the powerpc-elf toolset and is invoked via the `powerpc-elf-gdb` comand.

The powerpc-elf toolset (which targets 603-like processors) is used for the discussion in Appendix A [Bareboard and Custom Kernel Topics], page 7, and understanding the contents of that section is presumed in this Appendix, which focuses on a few 55xx-specific topics.

I.2 Floating Point

The processors targeted by GNAT for the PowerPC 55xx ELF incorporate an "Embedded Scalar FPU" that performs single precision floating-point operations on values in the general purpose registers. Relative to the 603, there are different instructions (`esfxxxx`), and there are no floating-point registers nor any of the 603 floating-point instructions. The version of the `libgcc` library provided with this toolset implements double-precision operations in software. The compiler will generate scalar FPU instructions when appropriate and calls to the double-precision routines when needed. Developers may want to use the scalar FPU support for efficiency, and/or avoid the library routines for certification. For objects and values of single-precision types, the hardware instructions will be used. However, some operations that are not simple floating-point operations may require usage of double-precision values, and therefore generate calls to to support routines in `libgcc`. For example, Ada semantics for rounding may force floating-point to integer conversions to use double precision.

For the developer who wants to eliminate or minimize the use of double precision, the issues are how to detect and how to work around such usage. If it is acceptable to eliminate support from `libgcc` entirely, then excluding it from the final linking step (e.g., using '`-nostdlib`' if linking with `gcc`, or simply not naming it if using `ld` directly), will result in error messages and failure to create an executable if any compiler-generated usage of double precision has occurred.

Then, to identify the source code involved, you can use `objdump` to inspect the object file that `ld` flagged as containing the unresolved reference. Invoke `objdump` (preceded by the prefix) using the '`--disassemble`' and '`--source`' switches. (Note: this requires that the code be compiled with debugging enabled.) The `objdump` command will produce a mixed source/assembly listing that can be searched for the undefined symbol named by `ld`.

If some minimal usage is acceptable, then various procedures are available to find new occurrences of double precision, such as occasionally linking without `libgcc` to look for new references, or inspecting object files with `nm`.

Eliminating a call to a double-precision routine depends upon the particular case and can involve modifying the source to use different constructs, adding qualifications or conversions, or providing machine-code insertions.

Notes:

- In order for `powerpc-eabispe-objdump` to disassemble the scalar floating-point instructions, the '`-M e500`' option is required.
- For CGNAT users: the '`-fsingle-precision-constant`' option to `gcc` will prevent floating-point constants from being implicitly converted to double precision.

I.3 EABI

The compiler is configured to conform with the EABI by default. However, we recommend avoiding the small data sections, both for simplicity and because (as of January 2008) they have not been as thoroughly exercised by user code as other product features. By default, gcc will place some data into `.sbss`, `.sbss2`, `.sdata` and `.sdata2`, and GPRs 2 and 13 will only be used for relative addressing as specified by the ABI and the EABI. Also, the compiler generates a call to `__eabi` before main, where the initialization of 2 and 13 can be made, and `libgcc` provides such a routine. By compiling with '`-mno-sdata`', no objects will be placed in any of the `.s`*XXX* sections and no use will be made of R2 and R13. This simplifies the construction of a linker script and allows the replacement of `__eabi` with these lines in the startup code:

```
    .global __eabi
  __eabi:
   blr
```

The provision of this dummy `__eabi` allows complete independence from `libgcc` for many programs.

I.4 Debugging

For bareboard 5554 targets, you can use debugging hardware that connects to the JTAG or Nexus port of the 5554 and communicates with GNAT's debugger by providing a `gdbserver` interface. Similar setups exist for many bareboard targets.

Appendix J AVR Topics

This Appendix describes topics that are specific to the GNAT for AVR cross configurations.

J.1 Introduction to GNAT for AVR

The AVR toolset targets AVR 8-Bit RISC microcontrollers from Atmel. AVR microcontrollers have limited resources: 32 8-bit wide registers, separate instructions and data memory, at most 256KB of flash for the instructions, and at most 8KB of internal SRAM for the data.

There is a large number of microcontrollers, and although they have the same instructions set core, they vary in capabilities. Some low-end models can be programmed only in assembly, while at the higher end one finds dedicated instructions to handle a larger amount of memory.

GNAT has been tested only with the atmega2560, with less support for the other microcontrollers.

J.2 Compiler and Linker Flags for AVR

The compiler must know which microcontroller is targeted, because code generation is affected. Use option '`-mmcu=ARCH`' where '`ARCH`' is the AVR architecture defined as follow:

- For '`atmega2560`' and '`atmega2561`' use '`-mmcu=avr6`'.
- For enhanced cores with 128KB of flash ('`atmega128x`' and '`at90usb128x`') use '`-mmcu=avr51`'.
- For enhanced cores whose flash size is between 8KB and 64KB use '`-mmcu=avr5`'.
- For enhanced cores with less than 8KB of flash use '`-mmcu=avr4`'.

Do not try to link object files compiled for different targets, since the code generated is not compatible.

The target should also be passed to the linker using the '`-mARCH`' option, since the linker may perform some optimizations or relocations on certain devices.

The general-purpose registers and the IO registers are mapped to data memory. As the number of IO registers is microcontroller specific, you must also set the start of the data. Use linker option '`-Tdata=0x00800NNN`' where '`NNN`' is the size of the IO registers in hexadecimal.

For example you can use this command to build program '`app`':

```
avr-gnatmake app -O -mmcu=avr6 -largs crt1.o -nostdlib -lgcc \
   -Wl,-mavr6,-Tdata=0x00800200
```

The option '`-O`' enables optimizations that may help reduce the code size.

The file '`crt1.o`' is the startup code. It contains the reset vector and interrupts table as well as initialization code to be executed before running compiled code. A sample startup code file is provided with the compiler installation.

The option '`-nostdlib`' prevents linking with standard libraries (such as the C library) which are not provided by GNAT, while '`-lgcc`' enables linking with the compiler support library. For some complex operations (such as 32-bit division) the compiler inserts calls to functions defined in this library.

J.3 Programming the AVR Chip

Use Atmel tools to programm the chips. The recommended method is to start from an '`ihex`' file which contains the image of the flash coded in hexadecimal. To convert from the output of the linker to the '`ihex`' format, use '`avr-objcopy`':

```
avr-objcopy -O ihex app app.ihex
```

J.4 AVR Registers Description

The file '`atmega2560.ads`', which is included with the AVR-specific examples, contains a package with a declaration for each IO register. With this package, you can directly access the IO registers without needing to declare them. Only the version for the '`ATmega2560`' is provided.

J.5 Writing Interrupt Handlers

It is possible to write an interrupt handler completely in Ada (i.e., with no assembly code). You have to add the '`signal`' machine attribute to your procedure so that it automatically saves and restores all registers used, and you also have to add a `pragma Export` to attach your procedure to the correct vector. The vector depends on the AVR target and is defined at the beginning of the '`crt`' startup file.

In the following example the procedure '`ADC_Interrupt`' will be called when an ADC interrupt is triggered:

```
procedure ADC_Interrupt;
pragma Machine_Attribute (ADC_Interrupt, "signal");
pragma Export (C, ADC_Interrupt, "__vector_adc");

procedure ADC_Interrupt is
begin
   -- ...
   -- Read current ADC value
   Data := ADC_Data;
   -- ...
end ADC_Interrupt;
```

J.6 Using the gdb Simulator

The easiest way to compile your program for the `gdb` simulator is to include the '`zfp_support.gpr`' project like this:

```
with "zfp_support.gpr";

project hello is
   for Source_Dirs use (".");
   for Main use ("hello.adb");
   package Compiler is
      for Default_Switches ("Ada") use ("-mmcu=avr6", "-g");
   end Compiler;
end hello;
```

The '`zfp_support.gpr`' project provides a small support library with a very simplified version of `Ada.Text_IO` and an appropriate link command. To compile a program, use the following command:

```
avr-gnatmake -P hello -XVARIANT=avr-avrtest
```

Once the binary is created, you can run it within `gdb`:

```
$  gdb hello

(gdb) target sim
(gdb) load
(gdb) run
```

J.7 Using Gdb With Atmel mkII Probe

The `gdb` debugger is able to use the Atmel JTAGICE mkII probe. Be sure to follow Atmel recommendations while using the probe (in particular, do not power-up a connected probe if the AVR micro-controller is not powered). Also do not forget to correctly program the fuse (be sure OCD and JTAG are enabled).

Contrary to the simulator, you can debug a real AVR micro-controller. But due to limitations in the AVR, you can only use 3 breakpoints. Note that for some operations (such as the `next` command), `gdb` uses temporary breakpoints. Also only connection through JTAG is supported.

To use the probe, you simply have to select the '`atmel-mkii`' target:

```
$ gdb myprog

(gdb) target atmel-mkii
(gdb) load
(gdb) run
```

As shown above, `gdb` is able to erase and program the AVR with the `load` command.

Appendix K LEON / ERC32 Topics

This chapter describes topics that are specific to the GNAT for LEON / ERC32 cross configurations.

K.1 Getting Started with GNAT for LEON / ERC32

This section describes topics that are relevant to GNAT for LEON / ERC32, a cross-development system supporting the Ravenscar tasking model on top of bare LEON / ERC32 computers.

ERC32 is a highly integrated, high-performance 32-bit RISC embedded processor implementing the SPARC architecture V7 specification. LEON is a 32-bit synthesizable processor core based on the SPARC V8 architecture. They have been developed with the support of the European Space Agency (ESA) as the current standard processors for spacecraft on-board computer systems.

The Ravenscar profile defines a subset of the tasking features of Ada which is amenable to static analysis for high integrity system certification, and that can be supported by a small, reliable run-time system. This profile is founded on state-of-the-art, deterministic concurrency constructs that define a model which has the required expressing power for constructing most types of real-time software.

The tasking model defined by the profile includes a fixed set of library level tasks and protected types and objects, a maximum of one protected entry with a simple boolean barrier for synchronization, a real-time clock, absolute delays, deterministic fixed-priority preemptive scheduling with ceiling locking access to protected objects, and protected procedure interrupt handlers, as well as some other features. Other features, such as dynamic tasks and protected objects, task entries, dynamic priorities, select statements, asynchronous transfer of control, relative delays, or calendar clock, are forbidden.

Using the cross-compilation system is very similar to its use in a native environment. The major difference is that the name of the cross tools are prefixed by the target name: `leon-elf-` / `erc32-elf-`.

```
$ leon-elf-gnatmake main
```

The result of this compilation is an ELF-32 SPARC executable. The tool chain has been tested to work on the following environments:

- A stand alone ERC32 (*TSC695F Rad-Hard 32-bit Embedded Processor*) computer based board.
- A stand alone LEON2-FT (*AT697E Rad-Hard 32-bit SPARC V8 Processor*) computer based board.
- A SIS ERC32 simulator (*SPARC Instruction Set Simulator*) running on the development workstation.

- TSIM ERC32 (*TSIM Simulator User's Manual*) running on the development workstation.
- TSIM LEON (*TSIM Simulator User's Manual*) running on the development workstation.
- QEMU (*QEMU Emulator User Documentation*) running on the development workstation.

The embedded multiplier/divider block in LEON2 AT697E can generate wrong values when negative operands are used (see *AT697E Errata Sheet*, erratum #1). The `-mcpu=v7` compiler switch must be used to avoid the generation of these potentially problematic instructions.

There is also a hardware problem of data dependency not properly checked in some double-precision FPU operations (see *AT697E Errata Sheet*, erratum #13) that affects LEON2 AT697E and AT697F boards. The `-mfix-at697f` compiler switch must be used to insert a `NOP` before the double-precision floating-point instruction.

A specific Board Support Package (BSP) is available for the Syderal ICM board using the `-micm` linker switch:

```
$ leon-elf-gnatmake -mcpu=v7 main -largs -micm
```

It is expected that other LEON / ERC32 boards, emulators or simulators could also be used with minor adaptations without problems.

K.2 Executing and Debugging on Emulators and Simulators

The program generated by the compilation toolchain can be executed using a LEON QEMU emulator on the development platform.

```
$ leon-elf-gnatmake main
$ leon-elf-qemu main
```

The same executable can also be run using the TSIM simulator.

```
$ tsim -freq 50 main
```

Typing `go` from the command prompt starts program execution.

There is another simulator for ERC32, called SIS, which supports the Motorola S-Record and not the ELF-32 SPARC executable format. Therefore, it requires an additional step to change the format of the executable.

```
$ erc32-elf-objcopy -O srec main main.srec
$ sis -freq 20 main.srec
```

Remote debugging using emulators/simulators is also possible. A remote debugger stub is needed in order to interact with `GDB`; this remote monitor can be either the one embedded into the emulators/simulator or the one provided with the GNAT development environment (`leon-stub` / `leon-stub-prom.srec`, `leon2-icm-stub` / `leon2-icm-stub-prom.srec`, `erc32-stub` / `erc32-stub-prom.srec`).

Both QEMU and TSIM embed a remote protocol for communicating with the debugger through an IP port (1234 by default) which is activated when using the -s switch on QEMU (the -S switch should be added to prevent the application from starting), and the -gdb switch or the gdb command on TSIM.

```
$ leon-elf-qemu main -s -S
```

The debugger can be launched directly from the command line or from GPS. Before loading the program to debug, GDB must connect to the simulator using the remote protocol and the required IP port.

```
$ leon-elf-gdb main
(gdb) target remote localhost:1234
   ...
(gdb) continue
   ...
(gdb) detach
   ...
```

K.3 Executing and Debugging on Hardware Platforms

The compiler generates ELF executables that can be converted into S-record files using the objcopy command (details are in the binutils documentation). :

```
$ leon-elf-objcopy -O srec main main.srec
$ grmon
grlib> load main.srec
grlib> run
```

Applications are linked to run from beginning of RAM, at address 0x40000000 for LEON (0x2000000 for ERC32). A boot-PROM can be created containing the application to be executed on a standalone target by using the leon-elf-mkprom / erc32-elf-mkprom utility. It will create a boot image that will initialize the board and memory, load the application into the beginning of RAM, and it will finally start the application. This utility will set all target dependent parameters, such as memory size, number of memory banks, waitstates, baudrate, and system clock. Applications do not set these parameters themselves, and thus do not need to be relinked for different board architectures.

The example below creates a boot-PROM for a system with 4 Mbyte RAM, 128 Kbyte ROM, a 50 MHz system clock, and UARTs programmed for 115200 baud rate. A file called main.srec is created, whose format is Motorola S-record, which is usually accepted by PROM recorders. It is possible to use an LEON / ERC32 simulator (SIS or TSIM) to load and test the resulting file.

```
$ leon-elf-mkprom -ramsize 4096 -romsize 128 -freq 50 -baud 115200 \
     main -o main.srec
```

Debugging software on the LEON / ERC32 board with GDB is possible by either using the provided GDB stub or using the remote target capability provided by GRMON.

A `GDB stub` is an embedded executable that provides the debugging API required by `GDB`, implementing the `GDB remote protocol` across a serial line.

The provided `GDB stubs` can be found at '`<install-dir>/(erc32|leon)-elf/lib`'. There are versions that can be loaded in memory and executed (`leon-stub` / `leon2-icm-stub` / `erc32-stub`), and others that can be copied into the boot-PROM (`leon-stub-prom.srec` / `leon2-icm-stub-prom.srec` / `erc32-stub-prom.srec`). At startup, this monitor installs itself into the upper 32K of RAM, and waits for `GDB` to be attached.

Remote target debugging requires a serial link between UART 2 (debugging link) on the remote target and a serial device on the host station. If the program produces some output to the console, another serial link connection is required from UART 1 (console on the target) to a terminal emulator on the host to display it.

The monitor supports break-in into a running program by pressing `Ctrl-C` in `GDB`, or interrupt in `GPS`. The two timers are stopped during monitor operation in order to preserve the notion of time for the application.

```
$ leon-elf-gdb main
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
   ...
(gdb) load
   ...
(gdb) continue
   ...
(gdb) detach
   ...
```

`GRMON` can act as a remote target for `GDB` allowing for symbolic debugging of target applications. This functionality is activated by launching the monitor with the `-gdb` switch or using the `GRMON gdb` command:

```
$ grmon -gdb
```

By default, `GRMON` listens on port 2222 for `GDB` connections:

```
$ leon-elf-gdb main
(gdb) set remotebaud 115200
(gdb) target remote :2222
   ...
(gdb) load
   ...
(gdb) continue
   ...
(gdb) detach
   ...
```

K.4 Adapting the Run-Time System

There may be some variations among the different boards, and therefore some mechanisms have been added to support adapting parameters easily without modifying the Ada run time.

The linker scripts, such as '(erc32|leon|leon2_icm).ld' that can be found at '<install-dir>/(erc32|leon)-elf/lib', define some basic board parameters, such as memory size and clock frequency. If these parameters need to be adapted, a local copy of this script can be created, and the following characteristics can be modified:

- Clock frequency. The clock frequency (in Hz) can be set modifying the value of the _CLOCK_SPEED variable to the desired value.
- Memory size. The _RAM_SIZE variable defines the size of the RAM memory installed in the board. There is also the memory map that needs to be tailored:

  ```
  MEMORY
  {
    ram (rwx) : ORIGIN = 0x40000000, LENGTH = New_Length
  }
  ```
- Stack size for the environment task. The stack area allocated to the environment task is defined by the variable _STACK_SIZE.

The local copy of the linker script can be selected from the command line using:

```
$ leon-elf-gnatmake main -largs -T my_script.ld
```

User-defined linker scripts can be used in two ways: users can either create a complete replacement of the default one, or modify only some of the values from the default linker script.

For example, if one needs to modify only the clock frequency, it makes sense to have a one-line user-defined linker script containing:

```
_CLOCK_SPEED = 50000000;
```

In this case, the goal is to merge the user-defined linker script together with the default linker script, and this is achieved using the -Tsmall_script.ld flag. Note the absence of a space between the -T and small_script.ld, which means that the specified linker script augments the default linker script.

```
$ leon-elf-gnatmake main -largs -Tsmall_script.ld
```

When the user-defined linker script is designed to completely replace the default linker script, this behavior can be obtained using the -T complete_script.ld option (note there is a space in it):

```
$ leon-elf-gnatmake main -largs -T complete_script.ld
```

With this command line, the default linker script provided by GNAT is completely ignored.

K.5 Run-Time Restrictions

The run time supports the tasking model defined by the Ravenscar profile; it has been specifically designed to take full advantage of this profile, that allows for a streamlined implementation of the Ada run-time library directly on top of bare LEON / ERC32 computer.

There are some other additional restrictions due to the bare board constraints, such as the removal of file system support, and complex text input-output. The exception handling mechanism that is implemented supports the full semantics of Ada 83 exceptions; Ada 95 enhancements are not included, but it supports propagation of exceptions and handlers for multiple tasks. It supports also limited Ada 95 exception occurrences, and `Ada.Exceptions.Exception_Name`. Mapping of the usual traps for hardware exceptions (division by zero, data access error, etc.) to Ada exceptions is also done.

In terms of Annexes supported, the status is the following:

- Annex A (*Predefined Language Environment*) is partially supported. Those functionalities related to file systems (A.7, A.8, A.14), complex text input-output (A.10, A.11, A.12), and command line access (A.15) are excluded.
- Annex B (*Interface to Other Languages*) is fully supported.
- Annex C (*System Programming*) is fully supported.
- Annex D (*Real-Time Systems*) is fully supported, excluding those features that are forbidden by the Ravenscar profile (D.4, D.5, D.11).
- Annex E (*Distributed Systems*) is fully excluded.
- Annex F (*Information Systems*) is partially supported. Those features related to text input-output (F.3.3, F.3.4) are excluded.
- Annex G (*Numerics*) is fully supported.
- Annex H (*Safety and Security*) is fully supported.
- Annex J (*Obsolescent Features*) is partially supported. Those functionalities related to text input-output, `Ada.Calendar`, and interrupt entries (J.7.1) are excluded.

K.6 Console Output

There is no console output support in the ZFP run-time library to avoid the inclusion of object code from this library. Ravenscar run-time libraries do provide this support.

The UART A on ERC32, or the UART 1 on LEON2 and LEON3, is used as the target console. When using the hardware board, a serial link connection is required from this UART port to a terminal emulator on the host (such as `tip`, `minicom`, `HyperTerminal`, etc.) to display it.

The Ravenscar profiles include the package `GNAT.IO` which can be used for displaying `Character`, `Integer`, and `String`.

```
with GNAT.IO;

procedure Main is
begin
   GNAT.IO.Put_Line ("Hello world");
end Main;
```

K.7 Stack Overflow Checking

GNAT does not perform stack overflow checking by default. This means that if the main environment task or some other task exceeds the available stack space, then unpredictable behavior will occur.

To activate stack checking, compile all units with the gcc option '`-fstack-check`'. For example:

```
$ leon-elf-gcc -c -fstack-check package1.adb
```

Units compiled with this option will generate extra instructions to check that any use of the stack (for procedure calls or for declaring local variables in declare blocks) do not exceed the available stack space. If the space is exceeded, then a `Storage_Error` exception is raised.

For declared tasks, the stack size is always controlled by the size given in an applicable `Storage_Size` pragma (or is set to the default size if no pragma is used). For the environment task, the stack size is defined by the linker script, and can be modified as described in Section K.4 [Adapting the Run-Time System], page 126.

K.8 Interrupt Handling

The Ravenscar profile allows only the use of protected procedures as interrupt handlers (as defined in Annex D of the *Ada Reference Manual*). Interrupt handlers are declared as parameterless protected procedures, attached to an interrupt source. All LEON / ERC32 interrupt sources are identified in package `Ada.Interrupts.Names` (including both name and priority associated). pragma `Attach_Handler` provides static attachment, and `Interrupt_Priority` establishes the priority of the handler.

```
protected Interrupt_Semaphore is
   pragma Interrupt_Priority
     (Ada.Interrupts.Names.External_Interrupt_2_Priority);

   entry Wait;

   procedure Signal;
   pragma Attach_Handler
```

```
            (Signal, Ada.Interrupts.Names.External_Interrupt_2);

      private
         Signaled : Boolean := False;
      end Interrupt_Semaphore;
```

The interrupt handler is executed on its own stack, at the priority given by the `Interrupt_Priority` pragma.

In order to reduce interrupt latency, the floating-point context is not automatically saved and restored when executing interrupt handlers. Floating-point computations are usually performed outside the protected handler, although if the handler is to execute floating-point instructions, the statements involved must save and restore the floating-point registers being used, in addition to enabling and disabling the floating-point unit.

An example of an interrupt handler that computes a square root using floating-point instructions is the following:

```
      with System.Machine_Code; use System.Machine_Code;

      procedure Handler is

         type Register_32 is mod 2 ** 32;
         for Register_32'Size use  32;

         PSR : Register_32;
         --  Processor State Register

         FSR : Register_32;
         --  Floating-point State Register

         F0  : Register_32;
         --  Floating-point register

         Operand : Float := 10.0;
         Result  : Float;

      begin
         --  First we enable the floating point unit, by means of setting
         --  the PSR's Enable FPU bit.

         Asm ("rd %%psr, %%l0" & ASCII.LF & ASCII.HT &
              "st %%l0, %0",
              Outputs => Register_32'Asm_Output ("=m", PSR),
              Clobber => "l0");

         PSR := PSR or 16#00001000#;

         Asm ("ld %0, %%l0" & ASCII.LF & ASCII.HT &
              "wr %%l0, %%psr",
              Inputs => Register_32'Asm_Input ("m", PSR),
              Clobber => "l0");

         --  Save the floating point registers that will be used (the
         --  Floating-point State Register and one of the Floating-point
         --  registers).

         Asm ("st %%fsr, %0",
              Outputs => Register_32'Asm_Output ("=m", FSR));

         Asm ("st %%f0, %0",
              Outputs => Register_32'Asm_Output ("=m", F0));

         --  User code (compute the square root)

         Asm ("ld %1, %%f0"       & ASCII.LF & ASCII.HT &
              "fsqrts %%f0, %%f0" & ASCII.LF & ASCII.HT &
              "st %%f0, %0",
              Inputs  => Float'Asm_Input  ("m", Operand),
              Outputs => Float'Asm_Output ("=m", Result),
              Clobber => "f0");

         --  Restore the floating-point registers previously saved

         Asm ("ld %0, %%fsr",
              Inputs => Register_32'Asm_Input ("m", FSR));

         Asm ("ld %0, %%f0",
              Inputs => Register_32'Asm_Input ("m", F0));

         --  Disable floating point operations (clearing
         --  the PSR's Enable FPU bit).

         PSR := PSR and not 16#00001000#;

         Asm ("ld %0, %%l0" & ASCII.LF & ASCII.HT &
              "wr %%l0, %%psr",
              Inputs => Register_32'Asm_Input ("m", PSR),
              Clobber => "l0");

      end Handler;
```

It is the user's responsibility to unmask the required interrupts in the Interrupt Mask Register. This way, the user has control over the point in time from which the interrupt will be acknowledged.

K.9 Non-Symbolic Traceback

A non-symbolic traceback is a list of addresses of call instructions. To enable this feature you must use the '`-E`' `gnatbind`'s option. With this option a stack traceback is stored as part of the exception occurrence.

Here is a simple example:

```
with GNAT.IO;
with Ada.Exceptions.Traceback;
with GNAT.Debug_Utilities;

procedure STB is

   procedure P1 is
      K : Positive := 1;
   begin
      K := K - 1;
   exception
      when E : others =>
         declare
            Buffer : constant Ada.Exceptions.Traceback.Tracebacks_Array :=
              Ada.Exceptions.Traceback.Tracebacks (E);
         begin
            GNAT.IO.Put_Line ("Call stack traceback locations:");

            for J in Buffer'Range loop
               GNAT.IO.Put (GNAT.Debug_Utilities.Image_C (Buffer (J)));
               GNAT.IO.Put (" ");
            end loop;
         end;
   end P1;

   procedure P2 is
   begin
      P1;
   end P2;

begin
   P2;
end STB;
```

```
$ leon-elf-gnatmake -g stb -bargs -E
$ leon-elf-qemu stb
Call stack traceback locations:
0x4000194C 0x40001518 0x400014F4 0x400014C0 0x4000116C
```

The location of these call instructions can be inspected with standard binary oriented tools, such as `nm` or `objdump`, or with the `addr2line` tool, that converts addresses into file names and line numbers. It is also possible to use `GDB` with these traceback addresses to debug the program. For example, we can break at a given code location, as reported in the stack traceback.

```
$ leon-elf-addr2line -e stb 0x4000194C 0x40001518 0x400014F4 \
     0x400014C0 0x4000116C

stb.adb:10
stb.adb:28
stb.adb:32
b~stb.adb:106
??:0

$ leon-elf-gdb stb
(gdb) break *0x4000194c
Breakpoint 1 at 0x4000194c: file stb.adb, line 10.
```

It is important to note that the stack traceback addresses do not change when debug information is included. This is particularly useful because it makes it possible to release software without debug information (to minimize object size), get a field report that includes a stack traceback whenever an internal bug occurs, and then be able to retrieve the sequence of calls with the same program compiled with debug information.

By default, unhandled exceptions display the stack traceback information stored within the exception occurrence.

Appendix L ELinOS Topics

This chapter describes topics that are specific to the GNAT for ELinOS, a cross-development Linux system.

L.1 Kernel Configuration for ELinOS

ELinOS comes with several tools to configure and build a linux kernel. For an extensive description of these tools, refers to the *ELinOS User's Guide*. Here are a few indications to help you configure your kernel in order to take full advantage of GNAT for ELinOS:

- *at project configuration*: First make sure that the kernel version that you are selecting is supported by GNAT for ELinOS. It is also advised, to be able to debug tasking applications on target, to have unstripped system libraries. To do so, you will have to switch off the checkbox `Tools⇒Configure mkefs⇒Strip symbol and debugging information`.
- *at feature configuration*: You should include the gdb server. It can be found in `Debugging⇒GDB server for remote debugging`.

L.2 Building a ELinOS Application

This section is starting point for building an Ada application targeting an ELinOS system.

To be able to build a program for ELinOS, the build toolchain will need to know where to find includes and libraries for this target. For that, the environment variable `ELINOS_CDK` and `ELINOS_TARGET` should be set. These are initialized the script `ELINOS.sh` that you may find at the root of your ELinOS kernel project.

To build the basic `Hello world` example on x86-elinos, one can then invoke `gnatmake` with the default options:

```
$ i686-elinos-linux-gnatmake hello
i686-elinos-linux-gcc -c hello.adb
i686-elinos-linux-gnatbind -x hello.ali
i686-elinos-linux-gnatlink hello.ali
```

L.3 Debugging an Application on ELinOS

This section contains the ELinOS-specific steps needed to run the ELinOS debugger.

The debugging solution that GNAT provides on this target consists in three tools:

- *gdbserver:* A debugging server on target; it provides an low-level interface to control the debugged program;

- *gdb:* The debugger itself; run on the host, it connects to the gdbserver and allows the user to control the debugged program through a command-line interface;
- *gps:* The GNAT Programming Studio; it provides a graphical front-end on the top of `gdb`.

To start a debugging session, you first need to program under the GDB server. Presuming the board has an internet connection to the development host, that the Internet hostname of the board is '`myboard`', that your application is name myapp and that the port on which the gdbserver will wait for requests is 2345, the command would be:

```
myboard> gdbserver myboard:2345 myapp
```

To be able to properly debug the program running on target, the debugger will need to know where to find a copy of the shared libraries that this program is executing. For that, the environment variable `ELINOS_PROJECT`, `ELINOS_CDK` and `ELINOS_TARGET` should be set. These are initialized the script `ELINOS.sh` that you may find at the root of your ELinOS kernel project.

You may now start the debugger. In GPS, on the `Languages` tab of the project properties page, you would specify the name of the debugger (e.g. `i686-elinos-linux-gdb`). You would then be able to start this debugger with the command: `Debug⇒Initialize⇒<No Main Program>`. Alternatively, you can run the debugger `gdb` from the command line.

You are now ready to connect the debugger to the GDB server:

```
(gdb) target remote myboard:2345
```

After the server has been started and attached from the host, the program is running on the target but has halted execution at the very beginning. You can then set breakpoints, resume the execution and use the usual debugger commands:

```
   ...
(gdb) break some_function
   ...
(gdb) continue
   ...
(gdb) next
   ...
```

For further information about the debugger or the GDB server, please refer to the *GDB User's Manual*.

Appendix M PikeOS Topics

This chapter describes topics that are specific to the GNAT for `PikeOS`, a cross-development ARINC-653 system provided by `SYSGO`.

M.1 Kernel Configuration for PikeOS

There are two possible ways to configure a `PikeOS` kernel:

1. `SYSGO`'s set of command-line tools: pikeos-cloneproject, pikeos-configure...
2. `CODEO`, a visually-oriented Integrated Development Environment based on the Eclipse framework.

In the following sections, we will explain how to use these tools to build a `PikeOS` kernel so that Ada applications can be run on it. The procedures will mostly use `CODEO`; to know more about `PikeOS` kernel configuration, refers to `PikeOS`' user manual *Installing and Using PikeOS*.

Two main settings are to be looked at when building a `PikeOS` kernel:

1. The partition maximum priority and the process maximum priority: Ada Ravenscar applications use up to 240 priority levels; in order to run properly, theirs partitions and processes should have a greater maximum priority. If this condition is not satisfied, the Ada application will exit at startup with error `MCP too low`.
2. In order to debug any application on target, a partition should run a tool called `muxa`. `muxa` provides several development services, such as a target console, and some debug channels that can be connected to AdaCore's debugger.

This will be illustrated by a simple scenario. The following instructions should help you build a simple kernel integration project in which Ada application can be run.

M.1.1 Creating a PikeOS kernel integration project

Make sure that you have all `PikeOS` tools in your `PATH`; then launch `CODEO`. In `CODEO`, create a new project by going to `File⇒New⇒Project...`.

This should open the `New project` wizard. Expand `CODEO⇒PikeOS` and select `Integration Project`. Click on `Next`. Give a project name and choose a workspace, then continue.

In the project templates, select `devel-pikeos`; it contains a project pre-configured with the major development features. Then click on `Finish`.

This will open the `PikeOS Project Configurator`; this is a visual editor for the configuration file '`project.xml.conf`'. In this editor, and for this scenario, select `qemu-ppc` in the `board` tab. `QEMU` is a simulator provided with `CODEO` that allows to easily run a `PikeOS` kernel on host; this will be useful in the context of this short tutorial. When this is done, click on `Save Settings`.

Note that you could also have a created this project from the command line with the command:

`pikeos-cloneproject /opt/pikeos-3.0/demos-integration/devel-pikeos new_project`

M.1.2 PikeOS maximum priority setting

You should now have a basic integration project; you can then configure it for Ada Ravenscar applications.

As mentioned previously, in order to run Ada tasking applications on a `PikeOS` process, you should make sure that this process can handle 240 priority levels. To do so, in the `Project Explorer`, double-click on '`vmit.xml`'. This will open the `Topology View`.

In this `Topology View`, in `PartitionTable`, expand `Partition -> pikeos⇒ProcessList`, then click on `ProcessEntry -> Proc_pikeos`. In its attribute, change `MaxPrio` to 240.

Then click on `Partition -> pikeos`. `CODEO` warns you about the fact that the process `Proc_pikeos` has a greater priority than its partitions; so upgrade the maximum priority for the partition as well. Set `MaxPrio` to 242. Then save your modifications.

M.1.3 PikeOS muxa configuration

As explained earlier, console and debugging functionnalities are provided by a component called `muxa` that comes in two parts: one running on target, one running on host. This section will focus on how to configure your kernel integration project in order to include this component on target. How to run the host part will be explained latter in this chapter (in section Section M.5 [Debugging an Ada application on PikeOS], page 142).

In the previously-created integration project in `CODEO`, double-click on '`project.xml.conf`' in the `Project Explorer`, to open the `PikeOS Project Configuration`. Expand `pikeos⇒service` and click on `muxa`. This will open a panel `Configuration Options: muxa`. Change the settings to match your host/target configuration. In our case, as we are building a kernel for `QEMU`, the following values will do:

- *mode:* networkfp
- *Device:* eth0:/0
- *HostIP:* IP address of your host;
- *HostPort:* port used by the host `muxa`; say, 1501;
- *TargetIP:* IP of the target as seen from target; say, 10.0.2.1 for `QEMU`;

- *TargetPort:* Port on which the target `muxa` is listening; say, 1500;
- *GatewayIP:* IP of the gateway from target; say, 10.0.2.2 for `QEMU`;
- *ConfigPort:* Port of `muxa`'s back office; say, 1550.

You can also change these settings from the command line, with the command

`pikeos-configure --editor`.

In order to have debug support for your particular user application, you will also need to make sure that it has access to the whole file system of the `muxa` partition. This is handled in the `Topology View`. To access to it, click on '`vmit.xml`' in the `Project Explorer`. Then, expand `Root -> Default⇒PartitionTable` and go your user partition: in this example, it is named `Partition -> pikeos`. Expand this node, right-click on `FileAccessList` and select `Add`. In the attributes of the newly created `FileAccess` node, set `muxa:*` for `FileName` and give it all access rights (Stat, Read. Write, Read-Write).

Your kernel integration project is now properly configured to host an Ada application. The following section will tell you how to build such an application and how to link it to the kernel.

M.2 Building an Ada application on PikeOS

This section will give a basic example to illustrate how an Ada Ravenscar application targeting `PikeOS` can be built. Consider the following Ada program, to be copied in a file named '`hello.adb`':

```
with GNAT.IO; use GNAT.IO;
with Ada.Real_Time; use Ada.Real_Time;

procedure Hello is
   Next : Time := Clock;
begin
   for J in 1 .. 5 loop
      delay until Next;
      Put_Line ("Hello from Ada!");
      Next := Next + Milliseconds (1_000);
   end loop;

   raise Program_Error;
end Hello;
```

This program prints `Hello from Ada!` five times, then exits on errors. With the ravenscar run-time library, only global exceptions are supported; and you have to provide a global handler in a Ada or C routine that exports `__gnat_last_chance_handler`. Here is an example of such a global handler, that you can save in a file named '`last_chance_handler.c`':

```
#include <string.h>
#include <stdio.h>
```

```
#include <vm.h>

void
__gnat_last_chance_handler (char *source_location, int line_number)
{
  static const char header [] = "Ada exception raised";
  static char msg [1024];

  if (strlen (header) + strlen (source_location) + 32 > 1024)
    {
      sprintf (msg, "%s (no location)", header);
    }
  else
    {
      sprintf (msg, "%s at %s:%d", header, source_location, line_number);
    }

  vm_hm_raise_error (VM_HM_EI_APP_ERROR,
     VM_HM_ERR_MSG_T_CUSTOM,
     msg, strlen (msg));
}
```

This example uses `PikeOS`' health monitor to report the error. To build this handler, do:

`powerpc-elf-pikeos-gcc -c -g last_chance_handler.c -I/opt/pikeos-3.0/target/ppc/oea/include/ -I/opt/pikeos-3.0/target/ppc/oea/include/stand/`

You can then build the Ada program and link it against this global handler with this command:

`powerpc-elf-pikeos-gnatmake -g hello.adb -largs last_chance_handler.o`

This will generate a file '`hello`'. You should then add this executable to your kernel integration project. Using the project created in the previous section Section M.1 [Kernel Configuration for PikeOS], page 137, here is how you would do: you would copy '`hello`' into the directory '`target/`', rename it to '`pikeos`' (as it is the name of the user application as specified in this project) and use `make boot` to link all pieces together:

```
$ cp hello <integration project dir>/target/pikeos
$ cd <integration project dir>
$ source ./PikeOS.sh
$ make boot
```

A file will be created: '`boot/pikeos-qemu-ppc-qemu`'; this is a `PikeOS` kernel running your simple Ada application. How to run it on `QEMU` will be explained in the next section.

M.3 Building an Ada application on PikeOS using GNATbench

This section will show how to use GNATbench to setup an Ada project similar to the one described in the previous section.

Click on File -> New -> Others and select Ada / Ada Project. Click on next. Give `demo` for the project name. Click on next. Give `hello` for the name of the Ada main subprogram, and check the first checkbox (generate the file). Click on next. Don't change anything for the directories settings, click on next. Check the third checkbox (Drive Build with a Makefile). Click on the scan button to refresh the names of the toolchains installed on the system, and select the powerpc-elf-pikeos toolchain (make sure that it's the only one checked). Hit finish.

A new project named demo should have been created. Open the file src/hello.adb, and paste the Ada code given in the previous section. Create a new file at the same location, last_chance_handler.c, and paste the corresponding code.

Open the file demo/Makefile. Change the build command to:

```
build:
powerpc-elf-pikeos-gcc -c -g  src/last_chance_handler.c \
-I/opt/pikeos-3.0/target/ppc/oea/include/ \
-I/opt/pikeos-3.0/target/ppc/oea/include/stand/ \
-o obj/last_chance_handler.o
$(GNATMAKE) -d -P "$(GPRPATH)" -largs obj/last_chance_handler.o
cp obj/hello ../kernel/target/pikeos
```

Right-click on the project demo, from the project navigator, and select `Build Current Project`. Then right-click on the target directory of the kernel project and select `Refresh`.

M.4 Running an Ada application on PikeOS

`PikeOS` and `CODEO` are integrated with a simulator called `QEMU`. This can be a simple way to run and test a `PikeOS` application.

In the previous sections, we have seen how to create a `PikeOS` kernel targeted to this simulator, embedding a simple Ada application. The following command should run this kernel on `QEMU`:

```
$ /opt/pikeos-3.0/share/qemu/bin/qemu-system-ppc -M prep_bare -serial stdio -nographic -
no-reboot -kernel boot/pikeos-qemu-ppc-qemu
PikeOS (C) Copyright 1998-2009 SYSGO AG, Germany
Build: S2151-3.0-443
Arch: PowerPC OEA
Platform: QEMU PPC PREP
Features: RETAIL TRACER-SYSCALL KERNEL-TRACING
PikeOS System Software starting. Build: S2151-3.0-151
PikeOS System Software up and running.
```

```
Hello from Ada!
PIKEOS_MON: Started, version: 3.0-131
Trace Server: version: 3.0-147
NE2K: mac addr 52:54:00:12:34:56 autodetected and assigned to Dev eth0:/0
NE2K: mac addr 01:04:9f:00:0a:49 assigned to virtual Dev eth0:/1
NE2K: mac addr 01:04:9f:00:0a:50 assigned to virtual Dev eth0:/2
NE2K: mac addr 01:04:9f:00:0a:51 assigned to virtual Dev eth0:/3
MUXA: version: 3.0-146
Hello from Ada!
Hello from Ada!
Hello from Ada!
Hello from Ada!
Hello from Ada!
ERROR <0001:006>: ERROR raised by Application.
ERROR <0001:006>: Message type: 1, size: 36
ERROR <0001:006>: Message content:
0000  41 64 61 20 65 78 63 65  70 74 69 6f 6e 20 72 61  |Ada exception ra|
0010  69 73 65 64 20 61 74 20  68 65 6c 6c 6f 2e 61 64  |ised at hello.ad|
0020  62 3a 31 33                                       |b:13|
ERROR <0001:006>: System State: PART_INIT
ERROR <0001:006>: Error Identifier: APP_ERROR
ERROR <0001:006>: Error Level: PL
ERROR <0001:006>: Action: ID
```

As expected, the message `Hello from Ada!` is printed and an exception is reported.

To quit QEMU, type Control-A Control-X.

M.5 Debugging an Ada application on PikeOS

Any application that should be debugged on PikeOS should call a routine named `init_gdbstub` at the early stages of its execution. To do so with GNAT, the simplest way will be to recompile the entry code of your Ada application. This entry point is a file named '`pikeos-app.c`' in the GNAT run-time library; the path to the GNAT run-time library is given by `powerpc-elf-pikeos-gnatls -v`. After copying it in your source directory, you can re-compile it with the following command:

```
powerpc-elf-pikeos-gcc -c -DDEBUG pikeos-app.c -I/opt/pikeos-3.0/target/ppc/oea/include/
```

Note the presence of an option `-DDEBUG`; it enables the gdbstub hook. After compiling it, you can link it to your application; in the case of our `hello` example, this build command would be:

```
powerpc-elf-pikeos-gnatmake -f -g hello.adb -largs pikeos-app.o last_chance_handler.o -lgdbstub -ldebug
```

Note that you should also link against '`libgdbstub.a`' and '`libdebug.a`' using the options `-lgdbstub -ldebug`. You can then re-copy it in the integration project that we created in the previous sections and re-link the kernel:

```
cp hello <integration project dir>/target/pikeos
cd <integration project dir>
source ./PikeOS.sh
make boot
```

To run this on `QEMU`, you shall specify several options to configure the network support properly, using the option `-net` and `-redir`; these options are documented in the usage message of `QEMU` that you can get by doing `qemu-system-ppc --help`. In our example, a proper setting would be:

```
/opt/pikeos-3.0/share/qemu/bin/qemu-system-ppc -M prep_bare -net nic -net user -redir udp:1500:10.0.2.1:1500 -serial stdio -nographic -no-reboot -kernel boot/pikeos-qemu-ppc-qemu
```

The next step is to launch the host `muxa`. Run it in your kernel integration project (replacing <*host*> by the actual name of your host machine):

```
muxa --type eth --host <host>:1550 --eth-host <host>:1501 --eth-target <host>:1500
```

You can then connect to it at <host>:1550 and configure your console and debug channels. For example, to get a console:

```
$ telnet <host> 1550
Ok: Connected to server.
muxa> lsn
Ok: Channel list, target supports 16 channels.
0  pikeos/Proc_pikeos/dbg
       Host:free      Port:0      Target:open     (Uid:0x000,0x0016,0x0000)
4  Monitor
       Host:free      Port:0      Target:open     (Uid:0x000,0x0004,0x0000)
5  mon_con
       Host:free      Port:0      Target:open     (Uid:0x000,0x0004,0x0008)
6  traceserver
       Host:free      Port:0      Target:open     (Uid:0x000,0x0005,0x0000)

muxa> connect mon_con
Ok: connected
help
PikeOS Monitor Command Shell v1.0
Available commands:
   help                 Help on command
   sysinfo              Info on system
   lsp                  List partition names
   pinfo                Info on a Partition
   lsc                  List channels
   cinfo                Info on a channel
   lsq                  List Queuing ports of a partition
```

```
      qinfo                 Info on a queuing port
      lss                   List sampling ports of a partition
      sinfo                 Info on a sampling port
      lse                   List processes of a partition
      einfo                 Info on a process
      lsm                   List memory requirements of a partition
      minfo                 Info on a memory requirement
      preboot               Reboot a Partition
      phalt                 Halts a Partition
      treset                Resets the target with the specified mode.
      kinfo                 Info on kernel structures
      ps                    Show Tasks/Threads

   Type ‘help name’ to find out more about the command ‘name’.
   PikeOS# lsp
    id name    partd fctid lctid mprio mode
     1 service   1.5     2    21   102 COLD_START
     2 pikeos    1.6    22    22   242 COLD_START
   PikeOS# pinfo pikeos
   operating mode:      COLD_START
   num_kpages:          8
   free_kpages:         3
   flags:               0
   max prio:            242
   part daemon:         1.6
   first child task:    22
   last child task:     22
   proc count:          1
   queuing port count:  0
   sampling port count: 0
   max. open files:     32
   open files:          1
   cookie:              0
```

...or to connect the application gdbstub to an host port, in order to debug your application:

```
   $ telnet <host> 1550
   Ok: Connected to server.
   muxa> lsn
   Ok: Channel list, target supports 16 channels.
   0  pikeos/Proc_pikeos/dbg
          Host:free      Port:0      Target:open     (Uid:0x000,0x0016,0x0000)
   4  Monitor
          Host:free      Port:0      Target:open     (Uid:0x000,0x0004,0x0000)
   5  mon_con
          Host:connected Port:1550   Target:open     (Uid:0x000,0x0004,0x0008)
   6  traceserver
          Host:free      Port:0      Target:open     (Uid:0x000,0x0005,0x0000)
```

```
muxa> connect 1551 pikeos/Proc_pikeos/dbg
Ok: Connected channel
```

You can then connect the debugger to <host>:1551 using `GDB`'s remote protocol, either from `GPS` or from the command line:

```
(gdb) target remote <host>:1551
Remote debugging using <host>:1551
0x08011d84 in gdbarch_breakpoint ()
(gdb) l hello.adb:2
1       with GNAT.IO; use GNAT.IO;
2       with Ada.Real_Time; use Ada.Real_Time;
3
4       procedure Hello is
5          Next : Time := Clock;
6       begin
7          for J in 1 .. 5 loop
8             delay until Next;
9             Put_Line ("Hello from Ada!");
10            Next := Next + Milliseconds (1_000);
(gdb) b 9
Breakpoint 1 at 0x80102c4: file hello.adb, line 9.
(gdb) c
Continuing.

Breakpoint 1, hello () at hello.adb:9
9             Put_Line ("Hello from Ada!");
(gdb) display J
1: J = 1
(gdb) c
Continuing.

Breakpoint 1, hello () at hello.adb:9
9             Put_Line ("Hello from Ada!");
1: J = 2
```

For further information about debugging with GNAT, refer to the *GNAT User's Guide* and to the *GDB User's Manual*.

M.6 Debugging an Ada application on PikeOS using GNATbench

The procedure that we just describe can also be integrated in the project that we create in section Section M.3 [Building an Ada application on PikeOS using GNATbench], page 140. To do so, the build target in '`demo/Makefile`' should now be:

```
build:
        powerpc-elf-pikeos-gcc -c -g  src/last_chance_handler.c \
        -I/opt/pikeos-3.0/target/ppc/oea/include/ \
        -I/opt/pikeos-3.0/target/ppc/oea/include/stand/ \
        -o obj/last_chance_handler.o
        powerpc-elf-pikeos-gcc -c -DDEBUG src/pikeos-app.c \
        -I/opt/pikeos-3.0/target/ppc/oea/include/ \
        -o obj/pikeos-app.o
        $(GNATMAKE) -f -d -P ‘‘$(GPRPATH)’’ \
        -largs obj/last_chance_handler.o obj/pikeos-app.o -lgdbstub -ldebug
        cp obj/hello ../kernel/target/pikeos
```

And copy ‘`pikeos-app.c`’ in directory ‘`demo/src`’. Then, rebuild this project (right-click on demo and `Build Current Project`).

You can then run `QEMU`, `muxa` and connect the debug channel to port 1551, as explained above.

Then, to run a debug session, right-click on your kernel project and select `Debug As`⇒`Debug Configurations...`. This will open a debug configuration named pikeos.

In this configuration, click on panel `Debugger`; set `GDB Debugger` to `powerpc-elf-pikeos-gdb`. Then click on tab `Connection`, select `TCP`, and set connection parameters to the debug channel as specify in muxa; here <host> and 1551.

Click on `Apply`, then `Debug`; the debug perspective will be opened in `CODEO` and the debugger will be attached to the target application. You can then navigate into your sources, set breakpoints in `hello.adb` by double-clicking on the line, inspect your local variables...

For further information about debugging from `CODEO`, consult the *CODEO User Guide* in `Help`⇒`Help Contents`.

M.7 Using the APEX Run-Time Library on PikeOS

GNAT for PikeOS provides two Ravenscar run-time libraries; one based on the native API, and the second one based on the APEX interface. The first part of this manual has been focused on the first one; this section will detail what should be changed to build, run and debug an application with the APEX Ravenscar run-time library.

M.7.1 PikeOS APEX Kernel Configuration

The same configuration as Section M.1 [Kernel Configuration for PikeOS], page 137 should work, with the following addition: an APEX partition will need an additional set of memory pools named `_APEX_KMEM_`, `_APEX_HEAP_POOL_`, `_APEX_STACK_POOL_`. To create them in `CODEO`, open vmit.xml and expand `Root -> Default`⇒`PartitionTable` and go your user partition: in our example, `Partition -> pikeos`. Under this node, right-click `MemoryRequirementList` and choose `Add`. You will have to do it for the three memory pools mentioned above; in each case, modify the newly added memory pool as follow:

- `_APEX_KMEM_`: change the name to `_APEX_KMEM_` and the type to `VM_MEM_TYPE_KMEM`; set the size so that the kernel has enough memory space to create the tasks of your application (e.g. `0x00200000`);
- `_APEX_HEAP_POOL_`: change the name to `_APEX_HEAP_POOL_` and the type to `VM_MEM_TYPE_RAM`; set the size to the heap size that your application needs to run properly (e.g. `0x00400000`);
- `_APEX_STACK_POOL_`: change the name to `_APEX_STACK_POOL_` and the type to `VM_MEM_TYPE_RAM`; set the size to the stack size that your application needs to run properly (e.g. `0x00400000`);

Then save your project and rebuild with `make boot` (from the command line or by a right-click on target `boot` in panel `Make targets`).

M.7.2 Building an Ada Application for PikeOS APEX

APEX gives the user the responsibility to start the scheduler after all its resources have been allocated; in our case, all tasks of the application. To do so in Ada, the proper way would be to perform all the “real” work in Ada tasks, and let the environment task start the scheduler. e.g. in the case of the hello example, it would mean to add a wrapper like this one:

```
with APEX_Wrapper_Tasking;

procedure APEX_Wrapper is
   procedure Set_Partition_Mode (Mode : Integer; Result : out Integer);
   pragma Import (C, Set_Partition_Mode, "SET_PARTITION_MODE");

   Set_Partition_Result : Integer;
begin
   Set_Partition_Mode (3, Set_Partition_Result);
end APEX_Wrapper;

--

package APEX_Wrapper_Tasking is
   task Env_Task;
end APEX_Wrapper_Tasking;

--

with Hello;

package body APEX_Wrapper_Tasking is

   task body Env_Task is
```

```
    begin
       Hello;
    end Env_Task;

 end APEX_Wrapper_Tasking;
```

In order to use this particular example of a wrapper, you may copy it into one file and use `powerpc-elf-pikeos-gnatchop` to break it into several Ada units:

```
 $ powerpc-elf-pikeos-gnatchop -w wr.ada

 splitting wr.ada into:
    apex_wrapper.adb
    apex_wrapper_tasking.ads
    apex_wrapper_tasking.adb
```

You can then use the same build procedure as in the previous section with the following differences:

- The main unit is now `apex_wrapper.adb` instead of `hello.adb`;
- The option `--RTS=ravenscar-apex` should be added to the options of `powerpc-elf-pikeos-gnatmake`;
- The options `-lapex` and `-lxt` should be added to the options of `gnatlink`; that is to say, it should be added in the `largs` section of the `gnatmake` command.

As an example, the build command given in Section M.2 [Building an Ada application on PikeOS], page 139 now becomes:

```
  powerpc-elf-pikeos-gnatmake -g --RTS=ravenscar-apex apex_
wrapper.adb -largs last_chance_handler.o -lapex -lxt
```

M.7.3 Debugging an Ada Application for PikeOS APEX

To debug an Ada Application using the APEX interface, the procedure is the same as the one described previously in sections Section M.5 [Debugging an Ada application on PikeOS], page 142 and Section M.6 [Debugging an Ada application on PikeOS using GNATbench], page 145 for the native interface. Report to these sections for a detailed example.

A set of differences is worth mentioning: the entry code to recompile should now be found in a file named '`pikeos-apex-app.c`' in the GNAT run-time library; the path to the GNAT run-time library is given by `powerpc-elf-pikeos-gnatls -v --RTS=ravenscar-apex`. After copying it in your source directory, you can re-compile it with the following command:

```
  powerpc-elf-pikeos-gcc -c -DDEBUG pikeos-apex-app.c -
I/opt/pikeos-3.0/target/ppc/oea/apex/include/
```

The debug libraries should be different as well; the link options `-lgdbstub`, `-lapex_d` and `lxt_d` should be added (in this order). In our example, the build command could be:

```
  powerpc-elf-pikeos-gnatmake -f --RTS=ravenscar-apex -g hello.adb
-largs pikeos-apex-app.o last_chance_handler.o -lgdbstub -lapex_d
-lxt_d
```

As a consequence, the Makefile fragment to be used in CODEO would be slightly different from the one given in section Section M.6 [Debugging an Ada application on PikeOS using GNATbench], page 145:

```
build:
        powerpc-elf-pikeos-gcc -c -g  src/last_chance_handler.c \
        -I/opt/pikeos-3.0/target/ppc/oea/include/ \
        -I/opt/pikeos-3.0/target/ppc/oea/include/stand/ \
        -o obj/last_chance_handler.o
        powerpc-elf-pikeos-gcc -c -DDEBUG src/pikeos-apex-app.c \
        -I/opt/pikeos-3.0/target/ppc/oea/include/ \
        -o obj/pikeos-apex-app.o
        $(GNATMAKE) --RTS=ravenscar-apex -f -d -P ``$(GPRPATH)'' \
        -largs obj/last_chance_handler.o obj/pikeos-apex-app.o \
        -lgdbstub -lapex_d -lxt_d
        cp obj/hello ../kernel/target/pikeos
```

Appendix N Customized Ravenscar Library Topics

This chapter describes how to use the GNAT cross tools to build programs that use the Ravenscar run-time library, and how to configure and port this library. It uses the PowerPC as the target platform to illustrate these issues.

N.1 Using project-based run-time libraries

The main advantage of project-based run-time libraries is that you can easily customize and modify them. However, the drawback is that they cannot be used as easily as the other standard pre-built run-time libraries; they can only be used through a project file and only with gprbuild.

If the application is built without a project file, or if it is being written from scratch, simply use this very simple project file '`prj.gpr`':

```
project Prj is
end Prj;
```

and build it with:

```
 gprbuild --target=powerpc-elf --RTS=<RTS> -P prj.gpr
```

where '`<RTS>`' is the root directory of the run-time library being used, which for the already-provided examples is a subdirectory of '`<gnat-root>/lib/gnat/<arch>/`'.

N.2 Porting the ZFP run-time library

This section describes how to customize the ZFP run-time library for a given hardware.

N.2.1 Sources organization

The sources and the binaries of the run-time library are provided in a hierarchy. Assuming that the base directory is '`<RTS>`', the following files and directories are present:

'`<RTS>/ada_object_path`'
'`<RTS>/ada_source_path`'

These files indicate where the object and source files of the run-time library are located. Each file simply contains a list of paths. You should not need to modify them (unless you want to add a new subdirectory).

'`<RTS>/common`'

This subdirectory contains source files that do not depend on the particular board. You should not need to modify them.

'`<RTS>/arch`'
: This subdirectory contains source files that depend on the board. You can edit them and add new source files (or remove some of them).

'`<RTS>/runtime.xml`'
: This is an XML configuration file for gprbuild. It mostly defines the compiler switches needed to compile user source files, and the linker script used. You may need to modify it if you want to port the kernel.

'`<RTS>/runtime_build.gpr`'
: This is the project file used to build the run-time library. As it inherits from '`<RTS>/runtime.gpr`', it is quite simple and you should not need to modify it. This project adds the '`-gnatg`' switch required to build run-time files.

'`<RTS>/lib`'
'`<RTS>/obj`'
: These subdirectory contains all the object files. They are automatically populated when the run-time library is built. You can remove their content to clean up the build. The '`obj/`' subdirectory contains object files that are built, while the '`lib/`' subdirectory contains the archive and ALI file that are used when an application is built.

N.2.2 Building the library

You can build the run-time library simply by using `gprbuild`:

```
$ gprbuild --target=powerpc-elf -P<RTS>/runtime_build.gpr
```

(You cannot build it with `gnatmake`, since the library is a multi-language project).

N.2.3 Board setup

The first step is to be able to run a very simple program that does not use any tasking features. This consists in initializing the board, the CPU, and the stack; possibly copying the data from flash to RAM; and calling `main`. This is not Ada specific, and it depends heavily on the target configuration. See Appendix A [Bareboard and Custom Kernel Topics], page 7.

You can start from the ZFP project-based example provided with GNAT. This project is located in '`<gnat-root>/lib/gnat/powerpc-elf/zfp-prep`' in the case of PowerPc. This example is for an emulator, so it might be simpler than for a real board (for example it is not necessary to initialize a DRAM controller or to test the RAM).

This example provides two subconfigurations: one ROM-based and one RAM-based. In order to use the RAM-based subconfiguration, you have to dynamically load and run the program using, for example, a ROM-based gdb stub. As a result, the RAM based one is slightly simpler.

To port the ROM-based subconfiguration, you need to modify the 'qemu-rom.ld' linker script (which is referenced only by '<RTS>/runtime.gpr'). You should also modify 'start-rom.S' which is the entry point (it defines the reset vector), set up the stack pointer, and copy data from ROM to RAM. This is where you can initialize the RAM if needed. This file also calls _setup (defined in 'setup.S') which simply clears the BSS segments. You should not need to modify 'setup.S'.

Then you can port Ada.Text_IO, whose body is in 'a-textio.adb'. The example drives a simple 16450-based UART.

Once done, a simple "hello world" program should work:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Hello is
begin
   Put_Line ("Hello zfp world");
end Hello;
```

N.3 Porting the GDB stub

Once the zfp run-time library is available, it may be worth porting the GDB stub, as it provides remote loading and debugging facilities. The GDB stub is a tiny zfp application that uses a serial line to communicate with a GDB hosted on the development machine.

On powerpc-elf the sources of the GDB stubs are in '<gnat-root>/share/examples/gnat-cross/gdbstub/powerpc-elf'. Two files have to be customized: the project file 'stub.gpr' must be modified to extend the zfp run-time library project for your board and the input/output file 'gdbstub_io.adb' which must be ported to your board.

Once the GDB stub is working, you also need to write a simplified BSP for the stub that doesn't initialize the board (as it is already done by the stub) and that maps the application in RAM.

N.4 Porting the Ravenscar run-time library

N.4.1 Overview

The Ravenscar run-time library is based on a very simple real-time kernel supporting preemptive fixed-priority scheduling. Tasks interact via protected objects, ensuring bounded priority inversion and absence of deadlock by using the priority ceiling locking policy (there are no explicit locks, but a task can block other tasks or interrupts by inheriting the ceiling priority while executing protected operations). Protected procedures can be used as statically bound interrupt handlers. See the Ravenscar Profile section in the *GNAT User's Guide Supplement for High-Integrity Edition Platforms* for more details about the Ravenscar tasking model.

It is assumed that you are able to write and execute simple programs that use the ZFP profile. It is also assumed that you have written a simple polling serial driver if you have a serial interface.

Most of the Ravenscar kernel for PowerPC is portable, as it depends almost entirely on features available on every PowerPC:

- It only handles one external interrupt. Although the kernel can be modified to handle more interrupts (discussed below), this basic scheme makes the porting effort easier.
- It uses the decrementer as the timer.
- The context saved is the one defined by the PowerPC UISA environment.

The following sections describe the parts that need to be adapted for a given PowerPC board.

N.4.2 Interrupts

The provided implementation of the Ravenscar run-time library for PowerPC uses only one interrupt priority for the external interrupt. Neither multiplexing interrupts nor attaching handlers are handled by this implementation. This simplifies the provided implementation and make it easy to port.

However, it is possible to extend the interrupt support in the run-time library. The first prerequisite is to know the number of interrupt priorities. You can then define the priority constants of package `System`.

The subtype `Any_Priority` represents the subtype for the priorities. This includes the priorities for normal tasks (subtype `Priority`) as well as the priorities for the interrupts (subtype `Interrupt_Priority`). As interrupts have a higher priority than normal tasks (because an interrupt will preempt a normal task), the `Interrupt_Priority` subtype values are just above the range for the `Priority` subtype.

It is possible to use a priority within the `Interrupt_Priority` range for a task, although this is not usual. In this case this task will block all the interrupts with a lower priority. This must be done with care, particularly if the clock or the timer are masked.

The range 0 .. 255 is recommended for the `Any_Priority` subtype, since 256 level of priorities should be enough, but you can change these bounds. The range does not affect the size of the run-time library, since there is currently only a single queue for the ready tasks. According to the ARM, `System.Priority` shall include at least 30 values.

You can always use one level of priority for interrupts, and all interrupts will be masked at that level. Note that this is the minimum value, as according to the ARM, the range of `System.Interrupt_Priority` shall include at least one value.

The range of `System.Interrupt_Priority` should map to the range of hardware priority levels, in order to have a one-to-one correspondence between them.

You must also define the `Default_Priority` constant which is the priority used for a task when it is not explicitly set by a `Pragma Priority`.

So ideally you just have to adjust `Nbr_Interrupt_Priorities` in '`system.ads`':

```
Nbr_Interrupt_Priorities : constant Positive := 32;
Max_Interrupt_Priority   : constant Positive := 255;

subtype Any_Priority       is Integer
  range   0 .. Max_Interrupt_Priority;
subtype Priority           is Any_Priority
  range   0 .. Max_Interrupt_Priority - Nbr_Interrupt_Priorities;
subtype Interrupt_Priority is Any_Priority
  range Priority'Last + 1 .. Max_Interrupt_Priority;

Default_Priority : constant Priority :=
  (Priority'Last - Priority'First) / 2;
```

You should not modify anything else in the '`system.ads`' file.

Once this is done, you can implement `Disable_Interrupts` and `Enable_Interrupts` in `System.BB.CPU_Primitives`. `Disable_Interrupts` is quite simple as it simply disables all interrupts, and the provided implementation should work. `Enable_Interrupts` has to enable all interrupts if the level is 0, and it has to disable all interrupt if the level is `Interrupt_Priority'Last`. In addition, `Enable_Interrupts` has to program the interrupt controller for levels between the two extremes of the range of interrupt levels (0 .. `Nbr_Interrupt_Priorities`). Of course things are simple if you only have one interrupt level.

You can also list the interrupt names and priorities in `Ada.Interrupt.Names`. That would be useful when you want to use interrupts.

You also have to define and implement the stack strategy. As the provided implementation has only one interrupt, it uses the application stack during the interrupt. But you could also have a stack dedicated to the execution of the interrupt, or even one stack per interrupt level.

N.4.3 Timer

Time is handled using the PowerPC decrementer and time base registers. Therefore it is very portable as it is available in the entire family of processors. The only implementation-defined parameter is the bus frequency, which also defines the frequency of the time registers.

The frequency is defined by the `Clock_Frequency` constant in the '`s-bbpara.ads`' file as in this example:

```
     Clock_Frequency : constant Positive := 66_000_000;
```

Appendix O GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related

2. VERBATIM COPYING

or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified

Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Heading 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify,

Index

A

addr2line 36
APEX (for VxWorks 653) 67
ARINC-653 67
ARINC-653 Binding (for VxWorks 653) . . . 80
`arincSchedSet` command (for VxWorks 653) 70
`Attach_Handler` pragma 129

B

Building from GPS 60
Building from Workbench 60

C

Conventions 3

D

Debugging (for VxWorks 653) 76

E

EABI (for VxWorks on PowerPc) 31
Embedded ABI (for VxWorks on PowerPc) 31
Exception handling (for VxWorks 653) 68

F

Free Documentation License, GNU 157
Full Ada Run-Time Library (for VxWorks 653) 67

G

`gdb` 12
`gdb` (for VxWorks 653) 76
GNAT VxWorks 653 Starter Kit 68
GNU Free Documentation License 157
GPS (for VxWorks 653) 76

H

Health Monitor (for VxWorks 653) 68

I

`Interrupt_Priority` pragma 129, 130

L

License, GNU Free Documentation 157

M

Mixed-language programming (for VxWorks) 17
Module OS (for VxWorks 653) 67

P

Partition OS (for VxWorks 653) 67
POSIX (for VxWorks 653) 67
PowerPc VxWorks, relocation issues 31
`pragma Discard_Names` 84
Project facility (in a cross environment) 5
Project files (for VxWorks 653) 69

R

Relocation issues for PowerPc VxWorks targets 31
Restricted Ravenscar Run-Time Library (for VxWorks 653) 67
Restricted Run-Time Library (for VxWorks 653) 67
RTPs and kernel modules 60

S

Secondary Stack 82

T

Tornado II Project 51
`torVars` script (for VxWorks 653) 77
Typographical conventions 3

V

vThreads (for VxWorks 653) 67
`vx653_system_build` 68
vxaddr2line 36
VxWorks 653 application structure 80
VxWorks 653 Topics 67
VxWorks kernel (relocation issues on PowerPc) 31
VxWorks MILS Topics 87
VxWorks PowerPc, relocation issues 31

W

Workbench / VxWorks 6 Topics 59

Z

Zero Footprint (ZFP) Run-Time Library (for VxWorks 653) 67

Table of Contents

About This Guide 1
What This Guide Contains 1
What You Should Know before Reading This Guide 2
Related Information 2
Conventions 3

1 Preliminary Note for Cross Platform Users 5

Appendix A Bareboard and Custom Kernel Topics 7
A.1 Introduction 7
A.2 Examples 7
A.3 Development Support 13

Appendix B Common VxWorks Topics 15
B.1 Executing a Program on VxWorks 15
B.1.1 Loading and Running the Program 15
B.1.2 Unloading the Program 16
B.2 Mixed-Language Programming 16
B.3 Kernel Configuration for VxWorks 17
B.4 Kernel Compilation Issues for VxWorks 18
B.5 Main Task Attributes and the Application Stub 20
B.6 Interrupt Handling for VxWorks 21
B.7 Handling Relocation Issues for PowerPc Targets 31
B.7.1 Background and Summary 31
B.7.2 Technical Details 31
B.8 Zero Cost Exceptions on PowerPC Targets 33
B.9 Calling exported Ada procedures from the VxWorks shell 33
B.10 Simulating Command Line Arguments for VxWorks 35
B.11 Using addr2line on VxWorks 36
B.11.1 Differences between VxWorks and native platforms 36
B.11.2 Using <target>-vxaddr2line 37
B.11.3 An example 37
B.12 Removal of Unused Code and Data 39
B.13 Debugging 39
B.14 Frequently Asked Questions for VxWorks 40

Appendix C Tornado 2.x / VxWorks 5.x Topics...... 43
C.1 Support for Software Floating Point on PowerPC Processors....... 43
C.2 Stack Overflow Checking on VxWorks 5.x.......................... 43
C.3 Debugging Issues for VxWorks 5.x................................ 44
C.3.1 Using the Debugger in Monotask Mode........................ 45
C.3.2 Using the Debugger in Multitask Mode....................... 47
C.3.3 Debugging an Ada Application Spawned from a C Program... 48
C.4 Using GNAT from the Tornado 2 Project Facility................... 51
C.4.1 GNAT as New Toolchain in the Tornado 2 Project Facility..... 51
C.4.2 Building a Simple Application................................ 52
C.4.3 Mixing C and Ada Code in a Tornado 2 Project................ 52
C.4.4 Compilation Switches.. 52
C.4.5 Autoscale and Minimal Kernel Configuration................. 53
C.4.6 Adapting BSPs to the GNAT toolchain........................ 53
C.4.7 Using GNAT Project Files in a Tornado 2 Project.............. 54
C.5 Using GNAT with the RTI ScopeTools............................. 55
C.5.1 General Information on Using the Tools...................... 56
C.5.1.1 Debug Switch... 56
C.5.1.2 Name Demangling..................................... 56
C.5.1.3 Symbol Downloading.................................. 57
C.5.2 MemScope.. 57
C.5.3 ProfileScope.. 57
C.5.4 TraceScope... 57
C.5.5 StethoScope.. 57

Appendix D Workbench / VxWorks 6.x Topics...... 59
D.1 GNAT for VxWorks 6 Installation................................ 59
D.2 Using GNAT for VxWorks 6....................................... 59
D.3 Building a VxWorks 6 Application................................ 60
D.3.1 Building from Workbench..................................... 60
D.3.2 Building from GPS... 60
D.3.3 RTPs and kernel modules..................................... 60
D.4 SMP Support.. 61
D.5 Using the VxWorks 6 Simulator.................................. 62
D.6 Debugging an Application on VxWorks 6.......................... 62
D.6.1 Using Workbench DFW.. 63
D.6.2 Using GPS and GDB.. 63

Appendix E VxWorks 653 Topics 67
E.1 Introduction 67
E.2 Setting Up a VxWorks 653 System 68
E.2.1 The GNAT VxWorks 653 Starter Kit 68
E.2.1.1 Configuration and Compilation Issues 68
E.2.1.2 Running the System 70
E.2.2 Manually Configuring a VxWorks 653 System 70
E.2.2.1 Application Configuration Files 70
E.2.2.2 Partition Makefiles 73
E.3 Running and Debugging Applications 76
E.3.1 VxWorks 653 System Setup 76
E.3.2 Environment Setup 77
E.3.3 GPS Setup 77
E.3.4 Debugging a Partition 77
E.3.5 Debugging Multiple Partitions 78
E.3.6 System Mode Debugging 78
E.3.7 Debugger Commands Specific to VxWorks 653 79
E.4 Application Design Considerations 79
E.4.1 General Application Structure on VxWorks 653 80
E.4.2 The ARINC-653 APEX Binding 80
E.4.2.1 Using the APEX Binding 82
E.4.2.2 Using Ada Processes 82
E.4.3 Selection of a Run-Time Profile 83
E.4.4 Replacement of the Default Last Chance Handler 84
E.4.5 Process-Level Health Monitoring Handler 85

Appendix F VxWorks MILS Topics 87
F.1 Introduction and Overview 87
F.2 Common Considerations for VxWorks MILS Applications 87
F.2.1 Configuration 87
F.2.2 GNATbench for MILS 88
F.2.3 Run-time Libraries for MILS 88
F.2.4 APEX Bindings 89
F.2.5 Multiplexed I/O 89
F.2.6 Communication between Virtual Boards 91
F.2.7 Debugging on MILS 92
F.2.8 Binding to MILS Header Files 92
F.3 Applications in the Guest OS Environment 92
F.3.1 Differences Between VxWorks 653 and VxWorks MILS vThreads Implementations 92
F.3.2 Ada Run-time Libraries for the Guest OS 93
F.3.3 Exceptions and Exception Handling on the Guest OS 94
F.3.4 Application Stub 94

F.3.5 Guest OS Virtual Board and Application Configuration Files . . 95
F.3.6 Setting Up a Guest OS Application with GNATbench 96
F.3.7 Guest OS Application Makefiles 96
F.4 Applications in the High Assurance Environment 99
F.4.1 Ada Run-time Libraries for the HAE 99
F.4.2 Exceptions and Exception Handling in the HAE 99
F.4.3 Security and Safety Critical Logs 100
F.4.4 HAE Virtual Board Configuration File 100
F.4.5 Setting Up an HAE Application with GNATbench 100
F.4.6 HAE Application Makefiles 101

Appendix G LynxOS Topics 105

G.1 Getting Started with GNAT on LynxOS 105
G.2 Kernel Configuration for LynxOS 105
G.3 Debugging Issues for LynxOS 105
G.3.1 Native Debugging on x86 LynxOS 106
G.3.2 Cross Debugging on PPC LynxOS 106
G.4 An Example Cross Debugging Session for LynxOS 107
G.5 Issues Linking on LynxOS 108

Appendix H AAMP Topics 111

H.1 Getting Started with GNAT for AAMP (GNAAMP) 111
H.2 GNAAMP-Specific Switches 113
H.3 Clock Support 114
H.4 Debugging for GNAAMP 114

Appendix I PowerPC 55xx ELF Topics 115

I.1 Introduction 115
I.2 Floating Point 115
I.3 EABI 116
I.4 Debugging 116

Appendix J AVR Topics 119

J.1 Introduction to GNAT for AVR 119
J.2 Compiler and Linker Flags for AVR 119
J.3 Programming the AVR Chip 120
J.4 AVR Registers Description 120
J.5 Writing Interrupt Handlers 120
J.6 Using the gdb Simulator 121
J.7 Using Gdb With Atmel mkII Probe 121

Appendix K LEON / ERC32 Topics 123
K.1 Getting Started with GNAT for LEON / ERC32 123
K.2 Executing and Debugging on Emulators and Simulators 124
K.3 Executing and Debugging on Hardware Platforms 125
K.4 Adapting the Run-Time System 126
K.5 Run-Time Restrictions 128
K.6 Console Output 128
K.7 Stack Overflow Checking 129
K.8 Interrupt Handling 129
K.9 Non-Symbolic Traceback 131

Appendix L ELinOS Topics 135
L.1 Kernel Configuration for ELinOS 135
L.2 Building a ELinOS Application 135
L.3 Debugging an Application on ELinOS 135

Appendix M PikeOS Topics 137
M.1 Kernel Configuration for PikeOS 137
M.1.1 Creating a PikeOS kernel integration project 137
M.1.2 PikeOS maximum priority setting 138
M.1.3 PikeOS muxa configuration 138
M.2 Building an Ada application on PikeOS 139
M.3 Building an Ada application on PikeOS using GNATbench 140
M.4 Running an Ada application on PikeOS 141
M.5 Debugging an Ada application on PikeOS 142
M.6 Debugging an Ada application on PikeOS using GNATbench 145
M.7 Using the APEX Run-Time Library on PikeOS 146
M.7.1 PikeOS APEX Kernel Configuration 146
M.7.2 Building an Ada Application for PikeOS APEX 147
M.7.3 Debugging an Ada Application for PikeOS APEX 148

Appendix N Customized Ravenscar Library Topics 151
N.1 Using project-based run-time libraries 151
N.2 Porting the ZFP run-time library 151
N.2.1 Sources organization 151
N.2.2 Building the library 152
N.2.3 Board setup 152
N.3 Porting the GDB stub 153
N.4 Porting the Ravenscar run-time library 153
N.4.1 Overview 153

N.4.2 Interrupts 154
N.4.3 Timer 155

Appendix O GNU Free Documentation License . . 157

Index 165