1

Calling Conventions

1.1 Introduction

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's **calling convention**. These rules address the following issues:

- *register preservation and usage*
- *parameter passing*
- *call and return mechanism*

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

This chapter describes the calling conventions used by the GNU C and standard HP-UX compilers for the PA-RISC architecture.

1.2 Processor Background

The PA-RISC architecture supports a simple yet effective call and return mechanism for subroutine calls where the caller and callee are both in the same address space. The compiler will not automatically generate subroutine calls which cross address spaces. A subroutine is invoked via the branch and link (**bl**) or the branch and link register (**blr**). These instructions save the return address in

a caller specified register. By convention, the return address is saved in **r2**. The callee is responsible for maintaining the return address so it can return to the correct address. The branch vectored (**bv**) instruction is used to branch to the return address and thus return from the subroutine to the caller. It is is important to note that the PA-RISC subroutine call and return mechanism does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

1.3 Calling Mechanism

All **RTEMS** directives are invoked as standard subroutines via a **bl** or a **blr** instuction with the return address assumed to be in **r2** and return to the user application via the **bv** instruction.

1.4 Register Usage

As discussed above, the **bl** and **blr** instructions do not automatically save any registers. **RTEMS** uses the registers **r1**, **r19** - **r26**, and **r31** as scratch registers. The PA-RISC calling convention specifies that the first four (4) arguments to subroutines are passed in registers **r23** - **r26**. After the arguments have been used, the contents of these registers may be altered. Register **r31** is the *millicode* scratch register. Millicode is the set of routines which support high-level languages on the PA-RISC by providing routines which are either too complex or too long for the compiler to generate inline code when these operations are needed. For example, indirect calls utilize a millicode routine. The scratch registers are not preserved by **RTEMS** directives therefore, the contents of these registers should not be assumed upon return from any **RTEMS** directive.

Surprisingly, when using the GNU C compiler at least integer multiplies are performed using the floating point registers. This is an important optimization because the PA-RISC does not have otherwise have hardware for multiplies. This has important ramifiications in regards to the PA-RISC port of **RTEMS**. On most processors, the floating point unit is ignored if the code only performs integer operations. This makes it easy for the application developer to predict whether or not any particular task will require floating point operations. This property is taken advantage of by **RTEMS** on other architectures to minimize the number of times the floating point context is saved and restored. However, on the PA-RISC architecture, **every** task is implicitly a floating point task. Additionally the state of the floating point unit must be saved and restored as part of the interrupt processing because for all practical purposes it is impossible to avoid the use of the floating point registers. It is unknown if the HP-UX C compiler shares this property.

- *NOTE: Later versions of the GNU C has a PA-RISC specific option to disable use of the floating point registers.* ***RTEMS*** *currently assumes that this option is not turned on. If the use of this option sets a built-in define, then it should be possible to modify the PA-RISC specific code such that all tasks are considered floating point only when this option is not used.*

1.5 Parameter Passing

RTEMS assumes that the first four (4) arguments are placed in the appropriate registers (**r26**, **r25**, **r24**, and **r23**) and, if needed, additional are placed on the current stack before the directive is invoked via the **bl** or **blr** instruction. The first argument is placed in **r26**, the second is placed in **r25**, and so forth. The following pseudo-code illustrates the typical sequence used to call a **RTEMS** directive with three (3) arguments:

```
set r24 to the third argument
set r25 to the second argument
set r26 to the first argument
invoke directive
```

The stack on the PA-RISC grows upward -- i.e. "pushing" onto the stack results in the address in the stack pointer becoming numerically larger. By convention, **r27** is used as the stack pointer. The standard stack frame consists of a minimum of sixty-four (64) bytes and is the responsibility of the callee to maintain.

1.6 User-Provided Routines

All user-provided routines invoked by **RTEMS**, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.