

Developing Embedded Applications

This chapter is intended to assist you in developing embedded applications for PowerPC targets. It contains the following sections:

§D.1: *PowerPC Models and Features Not Supported by High C/C++*

§D.2: *Getting Started: Hardware and Software Considerations*

§D.3: *Start-Up Code for Executables: An Example*

§D.4: *Storing crt1.o in an Archive Library*

§D.5: *Writing Start-Up Code for Shared Libraries*

§D.6: *Mapping Exception-Handling Code in Memory*

D.1 PowerPC Models and Features Not Supported by High C/C++

Currently, the High C/C++ compiler generates code only for the core PowerPC instruction set, as defined in the PowerPC architectural manual. High C/C++ code runs on any PowerPC model that implements the core PowerPC instruction set. The MetaWare ELF assembler supports instruction sets for PowerPC 403, 601, 602, 603 and 604.

On PowerPCs that lack a floating-point coprocessor, you must compile with option **-fsoft**. This option causes the compiler to generate calls to emulation routines that perform floating-point operations. These functions are described in the **PowerPC Embedded Application Binary Interface**.

D.2 Getting Started: Hardware and Software Considerations

This section presents steps for getting started in embedded programming on the PowerPC:

Step 1 Make sure you have adequate documentation about the PowerPC processor you are targeting.

Step 2 Consider your loader and host platform capabilities:

- Do they support shared libraries?
- Do they support shared libraries (DLLs)?
- Do they support virtual memory? The absence of virtual memory limits the way you implement shared library (DLL) support.
- Do they support multi-threading? If so, do you want to support multi-threaded C++ exception handling?
- What are the characteristics of your specific target platform? What input/output peripherals and memory constraints does it have? These factors affect your choice of a host platform and which I/O library functions you can use.
- Which function library do you plan to use? Do you plan to write your own, or use the generic version that MetaWare supplies for embedded development (similar to the library for Solaris on PowerPC)?

D.2.1 Function Libraries

Function libraries differ considerably from one operating environment to another. If you do not want to write your own library source code, you can use the generic version of the MetaWare library and supply the underlying system calls required (**read**, **write**, **open**, and so on).

D.3 Start-Up Code for Executables: An Example

The start-up module for an embedded PowerPC application is usually called `crt1.o`. You must write the source code for this file yourself. The following `crt1.s` serves as both an example and a guideline for writing your own start-up code for a PowerPC application written in High C/C++. You can modify it as needed to fit your system requirements.

A typical start-up module generally has the following parts:

- Entry point
- Stack setup
- System-specific initialization
- Calls to functions `main()` and `_exit()`
- DONE statement

Entry point By default, the linker uses the symbol `_start` as the application's entry point. You can specify a different entry point by using linker option `-e` on the linker command line, like this:

```
ldppc -e __entry_point
```

In this sample `crt1.s`, `_start` is assumed to be the entry point.

Stack set-up If your system does not have a loader that sets up the application's stack before passing control to the application, or if the application is responsible for setting up its own stack, use or modify the stack set-up code in `crt1.s`.

On most systems that allow dynamic expansion of the stack, the system's loader performs stack set-up. If your system has a loader that sets up the stack for your application, comment out the section in the code marked `STACK SET-UP`.

If your system has an operating system and you want to return control to the operating system when your application terminates, it is better to have the loader set up the application's stack. Otherwise, you must use the stack set-up code to save the system stack — a complicated process.

System-specific initialization After stack set-up and before the call to function `main()`, you must insert any other initialization code specific to your system. Anything that is not

handled by the loader has to be taken care of in the start-up code before control is passed to the application's function `main()`.

Calls to functions Function `_exit()` cleans up, then makes calls to the list of “at exit” functions added by function `atexit()`.
`main()` and
`_exit()`

DONE statement The `DONE` statement effectively halts the system when the application returns from function `main()`.

Here is the source code for `crt1.s`, for applications in both C and C++:

Example D.1 Source Code for `crt1.s`

```
! *****
! ENTRY POINT
! *****
        .section .text, text
_start:
        .globl _start
! *****
! STACK SET-UP:
! *****
! Allocate space for stack. Change the size of
! STACKSIZE as needed.
        .set      STACKSIZE, 4096
        .section .stack, bss
        .align    3
        .skip     STACKSIZE
STKTOP:
        .previous
! Initialize stack pointer.
        addis     %r11, %r0, STKTOP@h
        ori       %r1, %r11, STKTOP@l
        addi      %r0, %r0, 0
        stwu      %r0, %r0, 0
! *****
! SYSTEM-SPECIFIC INITIALIZATION
! *****
!
! Your system-specific initialization code goes here.
!
```

```

! *****
! CALLS TO FUNCTIONS main() AND _exit()
! *****
        bl        __init        ! Function __init() is
                                ! defined in crt1.s
        addis      %r3, %r0, __fini@h
        ori        %r3, %r3, __fini@l
        bl        atexit        ! atexit(&__fini())
        bl        main
        bl        _exit
! *****
! DONE STATEMENT
! *****
DONE:    b         DONE

```

Two object modules, `crti.o` and `crtn.o`, must be linked with your start-up module `crt1.o`, because they provide vital elements of your embedded application.

- Module `crti.o` provides the prolog code for the functions `__init()` and `__fini()`.
- Module `crtn.o` provides the epilog code for functions `__init()` and `__fini()`.

Function `__init()` resides in the `.init` section of your embedded application. Function `__fini()` resides in the `.fini` section.

The assembly source code for these files is contained in `crti.s` and `crtn.s`, which are located in the `lib/src` directory of the High C/C++ compiler distribution.

D.3.1 Link Order

The link order of `crti.o` and `crtn.o` is important: `crti.o` must be the first object module to be linked and `crtn.o` must be the last object module to be linked, after all other object and library files.

Link your files in this order:

```
ldppc crt1.o object_files library_files crtn.o
```

The standard ELF linker behavior is to concatenate, in the specified link order, the corresponding sections of each compilation unit. The compiler generates initialization code (function `__init()`) in the `.init` section of your application and termination code (function `__fini()`) in the `.fini` section, as a series of function-body instructions without a prolog and epilog. The prolog and epilog code must be provided by `crti.o` and `crti.o` respectively.

The compiler also generates code in the `.init` and `.fini` sections in the following situations:

- when you force code generation with `pragma Init` or `pragma Fini`
- when a C++ static object must be constructed before function `main()` executes
- when a C++ static object must be destructed after function `main()` terminates

In the MetaWare implementation of High C++, when a C++ static object is constructed or destructed, this code is placed in the `.init` section:

```
__mw_register_ctor(&constructor_item)
__mw_register_dtor(&destructor_item)
```

`constructor_item` and `destructor_item` are data structures with three fields:

```
.long initialization_priority_number
.long &function_to_construct_or_destroy_all_static_
    objects_in_module
.long link_field
```

Every module containing a static object that must be constructed or destructed has exactly one `constructor_item` and one `destructor_item`.

D.3.2 Setting Module Initialization Priority: `Pragma Initialization_level`

Function `__init()` calls the functions `__mw_register_ctor()` and `__mw_register_dtor()`:

- Function `__mw_register_ctor()` builds a priority queue of constructor items. At run time, when function `main()` starts to execute, it makes a call to function `_main()`, which calls `_mw_cpp_init()`. In turn, `__mw_cpp_init()` calls the constructor of each constructor item in the priority queue.
- Function `_mw_register_dtor()` builds a priority queue of destructor items. At run time, `__mw_cpp_exit()` is registered as an `atexit` function. After `main()` has terminated, `atexit()` calls `__mw_cpp_exit()`, which in turn calls the destructor of each destructor item in the priority queue.

The order of constructor items and destructor items in the priority queue is controlled by the `initialization_priority_number` field of the constructor item or destructor item.

You can specify the initialization priority number of an item with `pragma Initialization_level`. `Pragma Initialization_level` takes as an argument an initialization-level number from 0 (zero) to 255, where 255 indicates the highest priority and 0 (zero) the lowest. You can use this `pragma` in every module that contains a static object that must be constructed and which requires a priority number different from the default 0 (zero). If you do not specify an initialization level, the default is assigned.

The item with the highest initialization level is constructed first and destroyed last. For example, `cout` in the I/O stream library must be constructed before it can be used in any other code. Therefore, in the module that defines `cout`, `pragma Initialization_level` takes 255 as its argument.

D.3.3 Required Start-Up Operations

Your start-up code *must* perform the following operations in the order given:

1. Initialize the environment of your specific embedded system.
2. Call function `__init()`.
3. Register function `__fini()` as an `atexit` function:

```
atexit(&__fini());
```
4. Call function `_main()`.
5. Call exit code that calls any functions, including `__fini()`, registered as `atexit` functions.
6. Return cleanly to the system (the responsibility of function `__fini()`).

When you compile a module as part of a C++ program with `main()` as its entry point, the compiler inserts a call to function `_main()`, a C++ library function. `_main()` in turn calls function `__mw_cpp_init()`, which processes the priority queue of constructor items described in §D.3.2, and registers function `__mw_cpp_exit()` as an `atexit` function. For example:

```
_main() {  
    ...  
    __mw_cpp_init();  
    atexit(&__mw_cpp_exit());  
    ...  
}
```

Source code for `__mw_register_ctor()`, `__mw_register_dtor()`, and `__mw_cpp_init()` is in the `lib/src` directory of your compiler distribution.

When you compile a module containing function `main()` for a C program, the compiler does not automatically insert the call to `_main()`. Keep this in mind when you link a module containing a C `main()` with one or more C++ modules, because the C++ modules might contain static objects that need to

be constructed and destructed. In a case like this, you can do one of the following things:

- Put a call to `_main()` in function `main()`.
- Insert the call to `_main()` in your startup code before the call to `main()`, then modify the module `cppinit.cpp` so that the body of `_main()` is called only once. `cppinit.cpp` is located in the `lib/src` directory of the compiler distribution.

(In this case, if you were to use the same start-up code for a C++ program, the call to `_main()` in the C++ function `main()` would be redundant but harmless.)

D.3.4 Special Uses for Start-Up Code

The following are examples of real-life problems that you can resolve by providing the proper interfaces in your start-up code:

- If you are developing the applications for some kind of real-time operating system, you might have to set up or initialize or connect some interprocess communication (IPC) devices (shared memory, semaphores, messages, streams, sockets, and so forth) or direct-memory access (DMA) engines before the user process can run.

The start-up code is a good place to do all these system-level initializations. The start-up code can also call the routines that unlink these IPC utilities during termination.

- The start-up code can set up and destroy file descriptors that will be used by every program in the application.
- On an embedded board, you might need to do a lot of application-specific self checks before running the application. The start-up code is the logical place to put calls to self-check functions.
- If you have operating-system support for signals, traps, and exception handling, you can set up generic handlers for these features in the start-up code. User definitions can later override these handlers. You

can also use the start-up code to define default signal dispositions, which user applications can alter if necessary.

- The start-up code can initiate handlers for panic recovery and graceful error exits.
- You can use the start-up code to initialize strategies for process synchronizations. For example, if you are writing an application that might encounter dead locks waiting for resources, as happens in the case of many real-time applications, you can call timer-and-alarm functions in the start-up code that wake up or kill a process after a specific period of time.

D.4 Storing `crt1.o` in an Archive Library

You can store `crt1.o` in an archive library if you do one of the following:

- label the entry point `__start`
- use the `-e` linker switch to pass the name of the entry point to the linker

D.5 Writing Start-Up Code for Shared Libraries

If your application uses shared libraries, your start-up code will probably be different from that described in §D.3: *Start-Up Code for Executables: An Example*, because a shared library might have to be initialized for every instance of a process that uses it. To carry out this initialization, there must be an entry point in the shared library known to the loader. Because this entry point has a global binding, its name must be different from that of the entry point in the application program; otherwise, there will be a duplicate definition error.

What follows is one possible strategy for building a shared library to be used by an embedded application. In this example, shared library `shared.so` has a start-up module `scrt1.o` with an entry point `__shared_start()`. The entry point of `shared.so` itself is `__shared_main()`, and its exit point is `__shared_exit()`.

Step 1 Create an archive library `libso.a` (unshared) that contains all the routines your shared libraries use. In this archive library, include a module with a dummy `__shared_main()` entry point routine that does nothing.

Step 2 Link your shared library files in this order:

```
ldppc scrt1.o crti.o object_files libso.a crtn.o
```

This `shared.so` might or might not need `__shared_main()` to initialize it. If `shared.so` does not need to be initialized, there is no `__shared_main()` in any of its object files, and it picks up the dummy one from `libso.a`.

If `shared.so` does need to be initialized by its own `__shared_main()`, you must put `__shared_main()` in one of the library compilation units, then link the corresponding object file specifically on the command line. If the linker finds a valid `__shared_main()`, the dummy one in `libso.a` library is ignored.

At run time, the loader invokes `__shared_start()` in `scrt1.o`. (The loader knows to do this because you specified `-e __shared_start` at link time.)

`__shared_start()` in turn invokes `__shared_main()` for every process that uses the shared library.

The operations that `scrt1.o` performs are very similar to those for `crt1.o` for an executable:

1. Initialize the environment of your specific embedded system
2. Call function `__init()`.
3. Register function `__fini()` as an `atexit` function:

```
atexit(&__fini());
```
4. Call function `__shared_main()`.
5. When the process that uses the shared library terminates, the loader must call the clean-up code for the shared library (in this example, call the clean-up code `__shared_exit`). `__shared_exit` in turn calls the list of `atexit` functions registered by the shared library.

This simple example serves to illustrate that it is up to the system developer of a particular operating system to decide on a naming convention for shared library start-up code.

D.6 Mapping Exception-Handling Code in Memory

PowerPC exception handlers start at the address 0x100; each subsequent exception handler falls on an increment thereof that is a multiple of 100H.

Normally, exception handlers go into the data section with a small routine to copy each handler to the correct address (0x100, 0x200, ... 0x2000) at program start-up.

However, you can also define a separate section to be mapped to 0x100 and do the padding yourself between your exception handlers, to get them on the proper 100H boundary. For example:

```
.section my_exception_handler,"ax"
  ..LEH1:
  ...
  ! Your handler code for the handler at 0x100.
  ...
  ..LEH2:
.space  ..LEH1-..LEH2+0x100      ! pad to next 0x100.
  ...
  ! Your handler code for handler at 0x200.
  ...
  ..LEH3:
.space  ..LEH2-..LEH3+0x100      ! pad to next 0x100.
  ...
  ! Your handler code for handler at 0x300.
  ...
  ! And so on.
```