Software standards and RCE machine code

Files on disk

The output of the compiler is in Executable and Linkable Format (ELF). We use three types of standard ELF files. Unlinked compiler output files are of type Relocatable. The core containing RTEMS and run-time libraries is of type Executable, i.e., the format for main programs. Modules are stored as Shared Object files, i.e., shared library format. ELF has just the single format for both uses: modules loaded explicitly by the application such as Python extension modules as well as for libraries with one memory image shared among multiple processes. In RTEMS there are no processes and there is only one address space for main memory so the "shared" part of "shared object" is irrelevant; we just want the load-on-demand capability. Some non-ELF platforms such as MacOS X use different file formats for loadable modules and for shared objects so the two are not interchangeable.

A core image and a module both contain a single loadable segment, a piece of data meant to be loaded into a contiguous region of memory. In both cases the segment contains a special nested segment called the "dynamic section" which is used by the dynamic linker to find the dynamic symbol table for the core or module.

Application Binary Interface (ABI)

An ABI is a set of rules obeyed by the machine code. It usually covers how registers are used, how structures are laid out, data alignment, types of relocation used, how functions are called, stack layout and similar low-level properties of the code.

PowerPC

There are several widely used standard ABIs relevant to RCE code:

- UNIX SYSTEM V R4 ABI PowerPC Supplement
- PowerPC Embedded ABI (EABI)
- Itanium C++ ABI
- Itanium C++ exception handling

SYSVR4 vs. EABI

The SYSVR4 ABI covers the low-level code issues for UNIX application code. The EABI is a set of modified rules for PowerPC processors in embedded systems, which relaxes many requirements which either make no sense for such systems or which would consume too much memory, assumed to be in short supply. These two ABIs cover issues for C code; for C++ an adaptation of the ABI for Intel's Itanium platform is widely used as a model by many tool chains, including GCC et al. The EABI defines two levels of conformance, basic and extended, where the latter is mostly concerned with issues pertaining to shared object libraries.

RTEMS 4.9 and later is EABI compatible in the sense that hand-writtem assembly code doesn't trash any registers used for special purposes; its C code is compatible if you use the same compiler options to build RTEMS as you do to build the application.

By default GCC's generated code conforms to the SYSVR4 ABI but patched and configured as recommeded by the RTEMS developers it produces EABI-conformant code. In either case the C++ specific parts of the code use the Itanium ABI.

Small-data areas

The SYSVR4 ABI describes the use of a small-data area (code sections .sdata and .sbss). There the compiler will place static variables and constants X as long as sizeof(X) is not greater than a threshold given by the compiler's -G option; by default the threshold is eight bytes. These items are addressed by a signed 16-bit offset relative a base register (GPR 13) which can be done in a single load or store instruction. Normal access to a static datum requires at least one more instruction. The EABI defines a second such data area accessed using GPR 2 and holding only const non-volatile items (.sdata2, .sbss2); the original area based on GPR 13 is then reserved for non-const or volatile items.

Our GCC will by default generate only the single SVR4 small-data area but will not use GPR2 to access it. The compiler's treatement of the small-data areas is controlled by the -msdata command-line option as follows:

-msdata setting	Emit .sdata/.sbss?	Emit .sdata2/.sbss2?	Use GPR2/13 as base?
default	Same as "sysv"		
none	No	No	n/a
data	Yes	No	No
sysv	Yes	No	Yes

eabi	Yes	Yes	Yes

The setting of -sdata trumps -meabi or -mno-eabi.

If allowed GCC 4 will place typeinfo and vtable data in small-data areas.

ABI-related default options for our GCC

Source files will be compiled with the following options enabled. See the GCC 4 documentation for details.

```
All files:
-m32          -mbig                  -mbig-endian
-mbss-plt     -mdlmzb                -meabi
-mfp-in-toc   -msvr4-struct-return
-msdata=data  -G 8

Addtional for module files:
-mlongcall    -fvisibility=hidden    -no-pic
-shared
```

System and application startup

The EABI assumes that the application has a function main(int argc, char **argv, char** envp) which represents the entry point for the application and which the compiler and linker handle specially, generating and/or linking in extra code that sets up the execution environment. In our case startup code must handle system as well as application startup. There is no main() function because in general the compiler's extra generated code would not be correct for RTEMS. Such setup as is needed is split between RTEMS and the BSP. Part of that setup is loading the base registers for the small-data areas, zeroing uninitialized data areas and running any C++ static constructors.

For example, when compiling main() for an EABI-conforming platform GCC adds calls to the functions __eabi() and zero_bss() in the GCC runtime library. The former sets up the small-data base registers and then calls __init() which runs the static C++ constructors. The trouble with this is that those two jobs can't be done at the same time when starting up RTEMS; the small-data base registers must be set (assuming they're needed) before the bulk of the initialization is done since RTEMS and newlib are written mostly in C, but you can't run any C++ code until RTEMS and newlib are ready. Under RTEMS 4.10 the BSP is expected to call __eabi() and zero_bss() in bsp_start() with some jiggery-pokery in the specs file and linker script preventing the constructors from being run. That is delayed until before the first task (usually the init task) is run.

Modules

Our modules are packaged as ELF shared objects but don't conform to the EABI specifications for such objects. We don't need the features that would be used on Unix-like platforms to allow one copy of the object in memory to appear at different virtual addresses in different processes: position-independent code, the global offset table (GOT), the procedure linkage table (PLT) and the special types ELF relocations that go with them. RTEMS supports only one address space so we just need the ability to load a piece of code at an arbitrary location, fix up the internal addresses and look up external definitions in the core or in other modules, ignoring the vestigial PLT and GOT that the compiler produces when both -shared and -fno-pic are enabled.

Normally an ELF shared object is composed of several segments each of which is used for data of similar properties, e.g., read-only, executable, etc. The dynamic linker would then place each segment at a different location in memory in a region with compatible attributes. At present we instead put everything in a single segment, load the entire ELF file into a block of memory and perform the fix-up and binding there. This simplifies things but causes a problem with uninitialized data items since they are allocated no space in ELF files, there's basically just a count of the number of bytes. We get around this by using the objcopy utility to expand and zero-fill that part of the file (technically, by changing the .bss section type from NOBITS to PROGBITS). Once we have memory properties defined under RTEMS (via custom MMU and allocator libraries) we'll eventually revert to a more orthodox structure and handling for our modules.