Module preparation (RCE dynamic linker)

g++ options

-shared The linker will expect each module to have a "dynamic section" and a hash table for those symbols with global visibility.

-fno-pic [Req-Elf, Req-Reloc]. Although PIC saves on relocations (one per function instead of one per function call) it requires the dynamic linker to know the special relocation types needed for the Global Offset Table and the Procedure Linkage Table. The latter is especially tricky to get right. It's simpler to use non-PIC and just deal with the regular relocations.

The other usual reason to use PIC, that most of the code can be mapped into multiple address spaces at the same time, doesn't apply here since RTEMS only offers one global address space.

-nostdlib We supply our own module-init and module-fini code. Another reason not to use the standard init/fini code is that their source code was compiled with -fpic.

-nostdlib also prevents searching of the standard C++ libraries. Instead we'll want to search the symbol table of the RTEMS+newlib executable either at the time the module is produced or at run time. [Req-Purity]).

-fvisiblity=hidden A shared object actually has two symbol tables, the normal one and a special "dynamic" table. The latter is the one that is searched in order to resolve inter-object references. In the past pretty much every non-static symbol from every object code file used to make the shared object went into the dynamic table. Since GCC 4.0 the compiler recognizes the -fvisibility option which controls the default visibility of symbols. Using -fvisibility=hidden makes "hidden" the default. Hidden symbols don't go into the dynamic symbol table so one can dramatically reduce its size. To make sure that the symbols associated with a variable, function or class *do* make it into the dynamic symbol table you need to mark it with __attribute__((visibility ("default"))):

```
#define EXPORT __attribute__((visibility ("default")))

int EXPORT counter;

int EXPORT foo(int i) { ... };

class EXPORT bar { ... };
```

-fvisibility=hidden also hides out-of-line code generated for inline functions and other compiler-generated code.

-mlongcall By default GCC will compile function calls into branch instructions which use 26-bit PC-relative offsets. In general this won't be adequate for inter-module calls so the compiler must be instructed to make all function calls "long" calls. These use full 32-bit addresses at the cost of some speed.

ld options

At present static linking must be done by calling ld explicitly rather than relying on g++ to do it. The latter uses the linker option --eh-frame-hdr which causes ld to abort with an error about a "nonrepresentable section" in the output.

-shared and **-nostdlib** as for g++.

-z combreloc makes sure that all relocation table entries go into a single contiguous sub-region of the shared object, even if by mischance they end up in more than one output section. This option also causes the linker to sort the entries so that those reference the same symbol occur together. This allows the dynamic linker to use a one-entry symbol lookup cache.

-T *scriptname* to use the custom linker script.

-Map *filename* to make a link map.

-soname *modulename.major.minor.branch* to set the soname to the module's name with version numbers and development branch appended.

-o *modulename.major.minor.branch.so*

Each module that the one being constructed depends on should be named on the linker command line so that each of their sonames will be recorded in a DT_NEEDED entry in the dynamic section.

-fhash-style=sysv to suppress generation of a GNU-style hash table.

The layout of a relocatable module.

Each module is a dynamic object with two segments (and therefore two entries in the Program Header Table). One segment contains everything and is named ALL in the static linker script. The other, DYNAMIC, is nested within the first and contains the special "dynamic section" which contains the list of "needed" modules, a pointer to the relocation section, a pointer to the dynamic symbol table and other data specific to shared objects.

PowerPC uses only RELA-type relocations.

Normally the .bss section has no associated data in the ELF file. Use objcopy is used to inflate the .bss section to its full size with zeros:

```
powerpc-rtems-objcopy --set-section-flags .bss=alloc,contents,load,data _module-file_
```

Since each code section is allocated inside the ELF object [Req-Reloc] their offsets from the beginning of the object must have the proper alignment; the alignment of the beginning absolute address of the object must have the strictest alignment used for any offset (page boundary). The virtual address assignments inside the object look like this:

Type	Name	Description
ELF header		
Program Header Table		
section	rcework	Working storage for the dynamic linker.
section	.dynamic	The "dynamic section".
section	.hash	The hash table for the dynamic symbol table.
section	.dynsym	The dynamic symbol table.
section	.dynstr	The strings for the dynamic symbol table.
section	.rela	All relocations.
zero fill		Align to page boundary.
section	.text	Executable code, constant data.
zero fill		Align to page boundary.
section	.data	Variables with initial values.
section	.bss	Variables with no initial value specified.

Since virtual addresses assigned by the static linker begin at zero, the offset of each piece within the object up to the start of .bss is the same as the virtual address. That is to say there are no "holes" in virtual address space; each address has a byte of data assigned to it even if it's just filler.

The correspondence between offset and virtual address is important because each relocation in a shared object uses a virtual address to say where the calculated address is to be stored.

All the data above are assigned to a single segment of type LOAD. A second segment of type DYNAMIC is nested within the first and contains just .dynamic.

All the data up to .rela is automatically generated by the static linker ld. This section contains all the relocations that could not be carried out by ld. The RELA tag in the dynamic section points to this section.

The Small Data Area straddles the border between .data and .bss. The Global Offset Table followed by the .sdata2 and .sdata areas comes right at the end of .data while .sbss2 and .sbss come right at the start of .bss. The Small Data Area is referenced using 16-bit offsets from the value of *_GLOBAL_OFFSET_TABLE_* so we need at least a token GOT. We don't need a PLT, however.

The data described above are all that's needed to perform relocation. The static linker normally generates the following data at the end of the object but doesn't assign it any virtual addresses.

Type	Name	Description
section	.comment	Compiler version info, etc.
section	.shstrtab	String table used by the Section Header Table, contains section names.
section	.symtab	The ordinary symbol table.
section	.strtab	The string table used by .symtab.
Section Header Table		Describes each section.

Content of the *rcework* section

```
struct RceWork {
  int state;
  void (*moduleInit)();
  void (*moduleFinish)();
  bool (*isCompatible)(const char*modname, int major, int minor, const char *branch));
};
```

state is the current state of the module. The legal values are

- RAW (0) - The initial value set by the static linker. It signifies that the module has never been check by the dynamic linker.
- BAD_FORMAT (1) - The module ELF object failed the initial checks.
- UNRELOCATED (2) - The module passed initial checks but has not yet had local relocations applied.
- RELOCATED (3) - Local relocations have been applied and some imports may have been resolved.
- BOUND (4) - All imports have been resolved.
- INITED (5) - The module initialization code has been run. Other module code may now be called.
- FINISHED (6) - The module finish code has been run. The module can't be used any more.

moduleInit is a pointer to the module initialization code (normally the value of the local symbol `rce_modinit`.

moduleFinish is a pointer to the module finish code (normally the value of the local symbol `rce_modfinish`.

isCompatible is a pointer to the module compatibility function (normally the address of the local function `rce_isCompatible` supplied by the module writer). The module's ELF object has recorded in it the sonames of the other modules named on the command line of the static linker when the module is created. Each soname contains a module name, major version number, minor version number and branch name. If the module writer doesn't supply a compatibility function then the dynamic linker assumes that the called module in memory must be exactly the same as the one recorded. For example if module X.1.1.main is linked to Y.2.2.test when the module is built then X will require that it be bound with Y.2.2.test and no other version of Y at run time. If the module writers know that Y.2.0.test and Y.2.1.test may also be safely used by X.1.1.main then they can give X a compatibility function like this:

```
// Restrict the versions of modules for which this module is a client.
extern "C" {
  static bool rce_isCompatible(const string &modname, int majorv, int minorv, const string &branch)
  {
    // We'll accept any system core.
    if (modname == "*core*") return true;
    // We can use Y.2.0-2.test.
    if (modname == "Y") {
      return 2 == majorv &&
             0 <= minorv &&
             2 >= minorv &&
             branch == "test";
    }
    // We're certain that we use only the system core and Y.
    return false;
  }
}
```

For more elaborate dependency control some form of table lookup would probably be easiest to maintain.

This function is called after binding but before module initialization (it's no use initializing if some modules are incompatible). Therefore it should use only standard APIs that are not likely to change often, e.g., compiler C++ support and the C++ runtime library. It should *not* use module code or module data requiring run-time initialization, in its own or any other application module.