


# Developer Handbook - Coding

 This is an unapproved draft. It is currently in progress and under review.

## 6.1 Introduction

rce executables and libraries are coded in C++. Developers have access to the apis exposed by the rce core: rtems runtime (C), newlib (C), rce utilities (C++), and dynamic linker (C++).

This chapter covers:

- [Modules, runnable modules, and static libraries](#)
- [Coding rce modules](#)
- [Coding rce runnable modules](#)
- [Additional considerations for rce runnable modules](#)

Code examples are presented as clearly and concisely as possible; each sample contains just enough code to illustrate a particular concept, and generally omits `#include` directives and namespace declarations for standard libraries.

## 6.2 Modules, runnable modules, and static libraries

rce binary objects are shareables, built in much the same way as shareables are built on Linux or other platforms. However, rce development imposes a few special coding requirements, and it's worth defining terms before spelling these requirements out.

The rce uses elf (Embedded and Linking Format) as its native object code format. All rce system images support dynamic linking of elf objects. Chapter X provides more detail on the customized elf object format used by the rce.

In rce parlance, elf objects are known as **modules**. Library code can be packaged into shareable **library modules** (or simply modules). rce executables are known as **runnable modules**. A runnable module is executed in the context of an rtems task; a runnable module together with its execution environment can be termed an **rce task**. An initial rce task is spawned at the end of the boot sequence.

rce modules are built on a host system by linking any combination of other modules and/or traditional static libraries, with some restrictions noted in this section.

## 6.3 Coding rce modules

Coding shareables for the rce differs in a few ways from coding shareables on other platforms.

- The standard C++ library is not available. Much of this functionality is made available via newlib.
- rce modules are not compiled using position independent code (pic).
- Symbols are hidden by default to keep module size manageable. Symbols must be explicitly exported.

### The minimal rce library module

There are no mandatory symbols or restrictions on the structure of rce library modules.

### Include requirements

The following table lists the include directives needed to access features of the rce production and development system images.

#include directive	API	Availability
#include <rtems.h>	RTEMS runtime library. Required for access to RTEMS directives. See <a href="#">XXX</a> for API documentation.	production, development
#include <stdio> #include <string> #include <signal> ...	newlib C library. See <a href="#">XXX</a> for API documentation.	production, development
#include "rce/dynalink/Module.hh" #include "rce/dynalink/RunnableModule.hh" #include "rce/dynalink/Linker.hh"	RCE dynamic linker.	production, development
#include "rce/fci/Task.hh" #include "rce/fci/TaskMetaData.hh"	Flash container interface (FCI) for access to RCE configuration memory.	production, development

#include "rce/debug/Debug.hh" #include "rce/debug/Message.hh" #include "rce/debug/Print.hh"	RCE debugging message support.	production, development
"include "rce/shell/FsiFile.hh"	File system interface (FSI) for loading modules from NFS filesystems	development

## Module initialization

When dividing functionality among library modules, be aware that the order in which static constructors (and destructors) are called for exported classes can change depending on which modules have been loaded or unloaded using the dynamic linker.

After relocating a set of modules, the dynamic linker performs a lookup of global symbols, inferring a dependency graph based on symbol references.

When instructed to initialize the relocated modules, the linker uses this graph to defines the order in which it calls any static constructors. The opposite order is used when the linker calls destructors and drops modules.

## Exporting symbols

In the recommended build sequence described in Chapter X, symbols are hidden by default, and exported only when marked accordingly. This keeps the number of symbols recorded in a module's dynamic section to a minimum, and hence module size down.

Name collisions cause the linker to fail. A given symbol may be exported from at most one module in the set of modules known to the dynamic linker.

Define a macro in library modules to mark symbols for export. The appropriate macro is predefined in the slac rce repository make system and is automatically applied to module builds.

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

int EXPORT globalData(150);
void EXPORT globalFunction(intx) {returnx+1;}
class EXPORT GlobalClass { ... };

int localFunction(int y) {return y+2;}
```

Client modules use extern as for ordinary globals.

Header files for importing modules should omit the EXPORTs.

**Steve:** I've been mulling over this note and can't parse it. What should developers avoid here?

## Compatibility testing

At build time, gnu ld can be used to label a client module with a list of modules it has dependencies upon (see Chapter X). However, for each module in the dependency list, only one version (the soname: module name, major, minor, branch) is recorded. If the dynamic linker cannot locate this required module, linking fails.

On the rce, symbolic links and tools like ldconfig are not available for module version management. However, to adjust the set of compatible versions, implement the client module with an rce\_isCompatible function for the dynamic linker to call.

```
// Define 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)
}
```

The dynamic linker calls this function after binding but before module initialization (it's no use initializing if some modules are incompatible). Therefore, the function 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.

For each module that has a compatibility function the linker passes that function the version information for the core and for each module present.. If a module doesn't have a compatibility function then the linker assumes that:

- Any version of the core is compatible
- Any version of any module not on the "needed" list recorded by ld is compatible
- A module on the needed list must have exactly the version recorded in the list

In the following example, a client module implements a test for the presence of version 2.0.test, 2.1.test, or 2.2.test of module "Y". Note that the rce core is a module like any other; test compatibility with the core as illustrated in line 8.

```
extern "C" {
    static bool rce_isCompatible(const string &modname,
                                int majorv,
                                int minorv,
                                const string &branch)
    {
        // Accept any rce 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;
    }
}
```

## 6.4 Coding rce runnable modules

rce executables are modules that include an entry point symbol, `rce_appmain`.

Simple, single-threaded rce runnables will typically initialize debugging support, then direct the loading and linking of any required library modules, and finally enter their main processing loop. A more complex rce runnable acting as a dispatcher/controller might set up a pool of posix threads and/or load, link and launch other runnables from rce configuration memory.

The rtems task environment differs from the posix environment in some important ways:

- Each rtems task has its own standard streams (stdin, stdout, stderr) but all rtems tasks share global file descriptors

Most of the information on library modules in Section 6.3 applies, with exceptions noted below.

### The minimal rce runnable module

The code skeleton of a runnable module is shown below.

```
# include <rtems.h>

extern "C" {
    rtems_task rce_appmain(rtems_task_argument) {

    }
}
```

### Include requirements

Runnable modules have access to the same apis as library modules. See above.

### Runnable module initialization

At a minimum, rce runnables should set up debugging support, through standard or custom libraries, or with the rce debug and service packages.

```

#include <rtems.h>

#include "rce/debug/Debug.hh"
#include "rce/debug/Print.hh"
#include "rce/service/Exception.hh"

extern "C" {
    rtems_task rce_appmain(rtems_task_argument) {
        try {
            RceDebug::clearMessages();

            myApplication();

        }
        catch (RceSvc::Exception& e) {
            RceDebug::printv("*** rce exception %s", e.what());
        }
        catch (std::exception& e) {
            RceDebug::printv("*** c++ exception %s", e.what());
        }
    }
}

```

## Compatibility testing

Runnable modules can implement an `rce_isCompatible` function. See above.

## stdin, stdout, stderr

**Steve:** I think the following short explanation is right, but I don't have enough info to get to the meat - namely, how is a programmer supposed to write a utility app they can use to send commands to running tasks from the shell? As I look at the bsp and /libbsp/<rce/dummy\_console.c, it looks to me like an `rce_memory_write` is registered to /dev/console (allowing `Rce::Debug` outputs), but there is no callback in place to **read** from /dev/console (???). So even if you modify the shell to start a task that can pipe input and output between tcp socket and /dev/console, tasks aren't listening to stdin? Hmm, seems they do this for the existing apps w/o shell, but do you know how?...

Under rtems, each task is assigned its own input, output, and error streams, and all tasks share global file descriptors 0, 1, 2 for stdin, stdout, and stderr, respectively. All inputs and outputs are initially tied to device /dev/console. This convention holds for all tasks except two spawned by booting the rce development system: `tlnd` (the telnet daemon) and the rce shell (which is spawned from `tlnd`). Standard streams for the daemon and hence the shell are tied to the telnet tcp socket, rather than the console.

To provide communication with rce tasks via the shell, ???.

## Coding multi-module applications

Loading and linking modules at run time is a matter of retrieving the desired elf objects from remote or local storage, then calling the dynamic linker to relocate, bind, and initialize. A singleton instance of the linker, created at the end of the boot sequence, is available to all application code.

## Retrieving modules from storage

When the rce has been booted from a development system, use the File System Interface (fsi) library to read modules from a remote nfs filesystem. The filesystem must be mounted; see the rce shell command reference in Chapter Z.

**Steve:** This (and code further down) is kind of a mashup of the sample code from the workshop and Jim's `runTask` shell command code. It's likely wrong. Can you fix it up? In particular, does the `_fixup` method take care of the `NoDelete()` issues, or must users still defined `NoDelete()`?

```

#include <trl/memory>
using std::trl::shared_ptr;

#include "rce/dynalink/Module.hh"
using RCE::ELF::Module;

#include "rce/shell/FsiFile.hh"

shared_ptr<Module> getModuleFromFile(string path) {
    RCE::ELF::Module* elfMod;
    // read the filepath and construct the ELF module.
    try {
        RCE::FSI::Module mod(path.c_str());
        elfMod = mod.image();
        shared_ptr<Module> ptr(
            new(elfMod) Module,
            NoDelete());
        return ptr;
    }
    catch (RceSvc::Exception& e) {
        printf("Error in reading Module: %s\n", e.what());
    }
}

```

Under either a development or production system, use the Flash Container Interface (fci) package to read modules from rce configuration memory.

**Jim:** This is where we don't have any access to library modules (ELF::Module), only ELF::RunnableModules ?

```

#include <trl/memory>
using std::trl::shared_ptr;

#include "rce/dynalink/Module.hh"
using RCE::ELF::Module;

#include "rce/fci/Exception.hh"
#include "rce/fci/Data.hh"
#include "rce/fci/DataMetaData.hh"

shared_ptr<Module> getModuleFromContainer(int containername) {
    RCE::ELF::Module* elfMod;
    RCE::FCI::DataMetaData* md;
    // read the Data image from flash
    try {
        RCE::FCI::Name name(containername);
        RCE::FCI::Data data(name);
        md = data.metadata();
        elfMod = data.image();
        shared_ptr<Module> ptr(
            new(elfMod) Module,
            NoDelete());
        return ptr;
    }
    catch (RCE::FCI::Error& e) {
        printf("Error in reading Data from flash: %s\n", e.what());
    }
}

```

## Calling the dynamic linker

The following sample illustrates the basic sequence for accessing the linker, then relocating, binding, and initializing modules.

```

#include <tr1/memory>
using std::tr1::shared_ptr;

#include <rtems.h>

#include "rce/dynalink/Module.hh"
using RCE::ELF::Module;
#include "rce/dynalink/Linker.hh"
using RCE::Dynalink::Linker;

// Access the singleton
Linker &lnk(Linker::instance());

// At this point the Linker is only aware of this (the calling) module.
shared_ptr<Module> mod1(getModuleFromContainer(7));
lnk.relocate(mod1);
// Now it is aware of mod1 as well.

shared_ptr<Module> mod2(getModuleFromContainer(10));
lnk.relocate(mod2);

// Call bind to satisfy symbolic references;
// each module's rce_isCompatible function (if defined) is run
lnk.bind();

lnk.init();           // Run init code for mod1,2.

```

## 6.5 Additional considerations for rce runnable modules

### Using posix threads from a runnable module

TBW.

### Executing a runnable module from another runnable module

Loading and executing runnable modules dynamically is similar to loading library modules: the "parent" runnable retrieves the necessary elf object from local or remote storage, and invokes the dynamic linker for loading and linking. The parent calls the "child" module's run method, passing in rtems task configuration parameters, at which point a new rtems task is spawned and execution of the runnable begins at the child module's rce\_appmain routine.

Again, a singleton instance of the linker, created at the end of the boot sequence, is available to all application code.

### Retrieving runnable modules from storage

The process is essentially the same as that described above, except that the elf module is represented as an RCE::ELF::RunnableModule.

### Calling the dynamic linker

The following sample illustrates the basic sequence for accessing the linker, loading the runnable module from configuration memory, then running it.

```

#include <tr1/memory>
using std::tr1::shared_ptr;

#include <rtems.h>

#include "rce/dynalink/RunnableModule.hh"
using RCE::ELF::RunnableModule;
#include "rce/dynalink/Linker.hh"
using RCE::Dynalink::Linker;

// Access the singleton
Linker &lnk(Linker::instance());

shared_ptr<Module> mod1(getModuleFromContainer(7));

// The following method:
// * Performs dynamic relocation, binding, and initialization
//   of the runnable module
// * Creates and starts the rtems task
mod1->run("XTSK", // An "example task"
        100, // priority
        RTEMS_MINIMUM_STACKSIZE,
        RTEMS_DEFAULT_MODES,
        RTEMS_DEFAULT_ATTRIBUTES);

```

The example shows the simplest case, in which the runnable module has no external dependencies and is the only module loaded from rce configuraiton memory.