Target API (RCE dynamic linker)

- Using the linker from other RCE code
- Giving your application a "dynamic section"
- Terminology
- C++ API
  - rcelinker/dynalink/statusCodes.hh
  - rcelinker/dynalink/Linker.hh
  - Linker(const void *dynsect, int majorv, int minorv, const string &branch)
  - const void *coreDynsect() const
  - int coreMajorv() const
  - int coreMinorv() const
  - const string &coreBranch() const
  - static void setCoreInstance(Linker *)
  - static Linker &coreInstance()
  - ~Linker()
  - status relocate(const vector<shared_ptr<ELF> > &mods, bool droppable=true)
  - status Linker::bind()
  - status Linker::init()
  - status Linker::call(const string &funcSymbol) const
  - status lookup(const string &symbol, void *&value) const
  - status drop(const vector<shared_ptr<ELF> > &mods)
  - status Linker::drop()
  - status clear()
  - linkerstate Linker::state() const
- Linker state transitions
  - BADCORE
  - ERROR
  - NOTBOUND
  - BOUND
  - INITED

Using the linker from other RCE code

The dynamic linker follows the RCE source code conventions; to get the code from cvs check out module rcelinker inside an RCE release directory:

```
source /afs/slac/g/npa/setup/npa_49.csh
cd ~/myrelease
cvs checkout -d . rcelinker
```

You must then add rcelinker to the list of projects in your release directory's projects.mk file. Once that's done you can build the dynalink library using the standard RCE make process:

```
gmake rcelinker.dynalink.ppc-rtems-rce405
```

To use the library in one of your own projects refer to it as rcelinker/dynalink in make variables, e.g., if building an application called foo put something like this in the appropriate constituents.mk file:

```
tgtnames := foo
tgtsrcs_foo := etc.
tgtlibs_foo    := rcelinker/dynalink \
                  etc. \
                  etc.
```

In your source code you'll need to refer to two header files in rcelinker/dynalink:

1. statusCodes.hh
2. Linker.hh

In order to construct and instance of Linker you have to give it the address of the "dynamic section" of some ELF object (assumed to be the system core containing RTEMS and newlib). This is discussed in the next section.

Giving your application a "dynamic section"

At present the linking of standard RCE applications such as "console" is fully static (linker option -Bstatic). This means that the ELF object produced has only one symbol table which must be searched sequentially. For RTEMS there are several thousand symbols so a sequential search has poor average performance. If you change the LXXFLAGS in your flags.mk file to include -Bdynamic and -fexport-dynamic instead of -Bstatic your application will have a second "dynamic" symbol table which is indexed using a hash table. The GNU linker "ld" creates a special "dynamic section" in the ELF object which the RCE linker can use to find the hash table and the second symbol table. "ld" creates a symbol named _DYNAMIC whose value is the address of the dynamic section.

Dynamic linking produces some new sections which must be allocated space by the script used for "ld":

- .interp, points to the name of the GNU/Linux dynamic linker (which we don't use) in .dynstr.
- .plt, the Procedure Linkage Table.
- .dynbss, container for data copied by copy relocations.
- .dynsbss, like .dynbss but for "small" objects.
- .dynamic, which contains information on where to find the following sections.
- .dynsym, the dynamic symbol table.
- .dynstr, the string table used by .dynsym.
- .hash, the ELF hash table used when searching .dynsym.

There is one more change you have to make in order to have "ld" make a dynamic section: you have to search at least one shared library. It can be a dummy with no useful content but it has to be searched nevertheless. The reason is that the combination of options -Bdynamic and -fexport-dynamic is intended for the making of main programs that contain symbols they want to make accessible to shared libraries via the GNU/Linux dynamic linker; that linker expects to find a dynamic section and hash table.

Terminology

bind - To attempt to satisfy the imports of all known modules with exports from the same set of modules or from the core.

core - The body of run-time code containing RTEMS, newlib and the dynamic linker code. It's assumed to contain a dynamic symbol table just as a module does. Although the dynamic linker searches this symbol table for definitions needed by modules core the itself is never modified by the linker.

export - A symbol definition made in the core or in a module that is made available to all modules.

import - A reference to a symbol that is not defined in the module containing the reference, e.g., a call to an external function.

init - To run the initialization code of each module in the proper order.

known module - A module that has been given to the linker via its relocate() method and which has not yet been dropped.

module - An ELF shared object to be manipulated by the linker. A module has to be laid out in a way expected by the linker.

relocate - To "fix up" each module so that its code and data reflect the module's actual location in memory. Imports are not handled in this step.

system core - See core.

C++ API

This section describes the API visible to users of the dynamic linker on the target.

rcelinker/dynalink/statusCodes.hh

```
namespace RceDynalink {

  // Status codes for various operations.
  enum status {
    OK,

    // The linker has a fixed maximum of the number of known modules.
    TOO_MANY_MODULES,

    // An ELF object has some fault in its structure that prevented
    // the operation from being carried out.
    BAD_ELF_OBJECT,

    // Two module names (sans versions) are equal. Either a new module
    // has the same name as one already known or two new modules have
    // the same name.
    DUPLICATE_MODNAME,

    // At least one module still has symbolic references not satisfied
    // by any definition in one of the others.
    UNDEFINED_REFERENCES,

    // The same symbolic name is exported from multiple modules.
    DUPLICATE_DEFINITIONS,

    // A module listed as NEEDED by another module is not present.
    MISSING_NEEDED,

    // A module (or the core) used by some other module is, according to that user,
    // of an incompatible version.
    WRONG_VERSION,

    // There is at least one cycle in the inter-module dependency
    // graph; it is impossible to determine the correct order in which
    // to initialize modules.
    DEPENDENCY_CYCLES,

    // Something blew up when we tried to run a module's
    // initialization code.
    INIT_ERROR,

    // Something blew up when we tried to run a module's
    // finalization code.
    FINISH_ERROR,

    // There is no exported definition for the symbolic name given as argument.
    SYMBOL_NOT_FOUND,

    // The given module is not a member of the set of known modules.
    MODULE_NOT_FOUND,

    // A proposed drop operation would have left some modules without
    // modules that they depend on. This can happen if an undroppable
    // module depends on a droppable module.
    EVIL_DROP,

    // The operation's prerequisites have not been satisfied.
    TOO_SOON,
    TOO_LATE,

    // Something that "can't happen" did.
    INTERNAL_ERROR
    };
}
```

rcelinker/dynalink/Linker.hh

```
#include <tr1/memory>
#include <string>
#include <vector>

#include "rcelinker/dynalink/ELF.hh"
#include "rcelinker/dynalink/statusCodes.hh"


namespace RCE {
  namespace Dynalink {
    using std::string;
    using std::vector;

    using std::tr1::shared_ptr;

    enum LinkerState {
      // The initial state if the system core's dynamic section was
      // found to be badly formed. All operations except state() are
      // rejected with status code BAD_ELF_OBJECT. There are no known
      // modules.
      BADCORE,

      // All known modules (if any) have been relocated but not all are
      // bound or initialized.  This is the only state besides BADCORE
      // in which the set of known modules may be empty. This is the
      // normal initial state.
      NOTBOUND,

      // All known modules have been relocated and bound but not all are
      // initialized.
      BOUND,

      // All known modules have been relocated, bound and initialized.
      INITED,

      // An internal error was detected and the Linker is useless.
      // All operations that return a status return INTERNAL_ERROR.
      ERROR
    };

    // The dynamic linker. Many instances may be created but
    // they are not copyable or assignable.
    class Linker {

    public:

      // Construct a new instance given information about the system core.
      //
      // dynsect - A pointer to the beginning of the dynamic section. If
      // you're calling the constructor from the core itself then the
      // value of the symbol _DYNAMIC_ is the correct value to use here.
      //
      // majorv, minorv - The major and minor version numbers.
      //
      // branch - The branch of developent, e.g., "main" or "test".
      //
      // The system core is never altered in any way nor is it ever
      // dropped.  The dynamic section is used to find the core's symbol
      // table and the hash table that goes with it. The version and
      // branch information is passed to module compatibility functions
      // along with the reserved name "*core*".
      //
      // State after call: NOTBOUND or BADCORE.
      Linker(const void *dynsect, int majorv, int minorv, const string &branch);

      const void    *coreDynsect() const;
      int            coreMajorv()  const;
      int            coreMinorv()  const;
      const string  &coreBranch()  const;
```

```
// Find the instance that was created in the core.
static void setCoreInstance(Linker *);
static Linker &coreInstance();

// Perform clear() then self-destruct.
~Linker();

// In addition to the state transition described below an
// operation may detect an internal Linker error in which case it
// returns INTERNAL_ERROR and enters the ERROR state.

// Add the ELF objects pointed to by the entries of mods to the
// set of known modules. The program instruction and program data
// areas will have relocations applied to them but undefined
// symbolic references will not be resolved at this time (see
// bind()).
//
// Modules added with droppable=false are not dropped by the drop()
// methods but are dropped by clear().
// The possible return codes are:
// OK                -> NOTBOUND
// TOO_MANY_MODULES  -> current state
// BAD_ELF_OBJECT    -> current state
// DUPLICATE_MODNAME -> current state
// DUPLICATE_DEFINITIONS -> current state
status relocate(const vector<shared_ptr<ELF> > &mods, bool droppable=true);

// Ensure that as far as possible all symbolic references between
// modules and from modules to system core have been resolved.
// Possible return codes:
// OK                    -> BOUND
// BAD_ELF_OBJECT        -> current state
// UNDEFINED_REFERENCES  -> current state
status bind();

// If all modules are bound, run the initialization code in each
// module (unless this has already been done). This requires a
// topological sort of modules by dependency. Module A depends on
// module B if A uses a symbol defined by B or if B is mentioned
// on the ld command line when module A is made. Possible return
// codes:
// OK             -> INITED
// BAD_ELF_OBJECT -> current state
// DEPENDENCY_CYCLES -> current state
// INIT_ERROR     -> current state
status init();

// If all modules are initialized, transfer control to the
// function named by the given symbol (which some module or the
// core must have exported). We recommend that you either create a
// new task that uses call() or else have the invoked function
// start a new task and then return.
// Possible return codes:
// OK               -> current state
// BAD_ELF_OBJECT   -> current state
// SYMBOL_NOT_FOUND -> current state
// TOO_SOON         -> current state
status call(const string &funcSymbol) const;

// Place in value the value of a symbol exported either by the core or by a module.
// OK               -> current state.
// BAD_ELF_OBJECT   -> current state.
// SYMBOL_NOT_FOUND -> current state.
status lookup(const string &symbol, void *&value) const;

// The ELF objects in mods should have been given earlier to
// relocate() and not yet dropped.  The corresponding modules
// that are droppable are finalized and references to them are
// removed from the linker's internal tables. Even if a module
// is not in mods, if it is droppable and any of the the modules
```

```
      // it depends on are dropped, the dependent module is also
      // dropped.
      // Possible return codes:
      // OK               -> current state, or NOTBOUND if no modules remain.
      // BAD_ELF_OBJECT   -> current state
      // MODULE_NOT_FOUND -> current state
      // EVIL_DROP        -> current state
      status drop(const vector<shared_ptr<ELF> > &mods);

      // Drop all droppable modules. It can return all the status codes
      // of drop(const vector<shared_ptr<ELF> >&) except MODULE_NOT_FOUND.
      status drop();

      // Like drop() except that the droppable flags are ignored; every
      // module gets dropped, returning the linker to its pristine
      // state.
      status clear();

      // Return the current state of this instance.
      LinkerState state() const;
    };
  }
}
```

Linker(const void *dynsect, int majorv, int minorv, const string &branch)

Construct a new instance given information about the system core.

dynsect A pointer to the beginning of the dynamic section. If you're calling the constructor from the core itself then the address of the symbol `_DYNAMIC`, i. e., `&_DYNAMIC` is the correct value to use here.

majorv, minorv The major and minor version numbers.

branch The branch of developent, e.g., "main" or "test".

The system core is never altered in any way nor is it ever dropped. The dynamic section is used to find the core's symbol table and the hash table that goes with it. The version and branch information is passed to module compatibility functions along with the reserved name "*core*".

State after call: NOTBOUND, or BADCORE if the dynamic section doesn't look right.

const void *coreDynsect() const

Returns the value of *dynsect* given to the constructor.

int coreMajorv() const

Returns the value of *majorv* given to the constructor.

int coreMinorv() const

Returns the value of *minorv* given to the constructor.

const string &coreBranch() const

Returns the value of *branch* given to the constructor.

static void setCoreInstance(Linker *)

Used by the core to save the location of the Linker instance it uses.

static Linker &coreInstance()

Used by application modules to retrieve a reference to the core's instance of Linker.

~Linker()

Destructor.

status relocate(const vector<shared_ptr<ELF> > &mods, bool droppable=true)

Accepts any number modules already in memory and relocates them in place. Relocated objects are added to set of known modules. The relocations performed are those that do not involve symbols marked as "undefined" which can be done for each module independently of all others.

No modules will have been added to the known set if the return code is not OK. It's possible that relocation will have been started on some of the new modules but it doesn't matter; such modules may still be resubmitted later.

No two modules are allowed to have the same base name, i.e., the full name with the version numbers and branch removed.

No two modules are allowed to define a value for the same global symbol, except for weak definitions which are always treated as strictly local.

relocate() may be called in between other operations in order to add to the current set of Modules; doing so will make it necessary to redo bind() and init().

Modules added with the droppable flag set to false will be unaffected by calls to drop() but may be dropped by using clear(). See the notes for Req-Purity.

status Linker::bind()

Attempts to resolve all symbolic references among the set of current set of relocated modules (no lazy binding). It is a non-fatal error if some symbolic references are left unresolved at the end of a call to bind(); one can try again after adding more modules with relocate().

During binding the linker records which modules depend on which so that init() may determine the right order in which to initialize them.

status Linker::init()

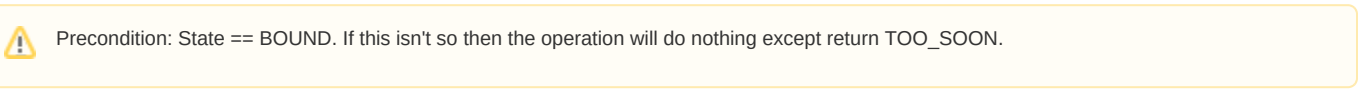
The eventual aim of this function is to run the initialization code for each known module in the correct order.

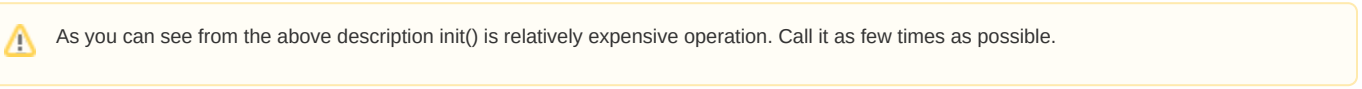
First, the linker looks for any explicit dependencies recorded in the module by the static linker ld. If when building module A you list module B on the ld command line then ld records that A needs B (the full "soname" of B is recorded). This is handy if there are dependencies that can't be discovered by looking at symbolic references. For example module B may install a device driver neeed by A; A and B don't refer to each other but only to the core.

For each module that has a compatibility function the linker passes that function the version information for the core and for each module present. The function returns true if the given version is deemed compatible. If a module doesn't have a compatibility function then the linker assumes that:

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

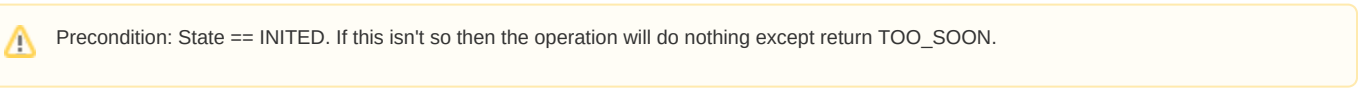
Once it has calculated and checked dependencies the linker performs a topological sort to determine the order of initialization, such that if module A depends on module B then B is sorted before A. If there are no dependency cycles then it calls each module's initialization function in the sorted order. No initialization calls are made if there are cycles.





status Linker::call(const string &funcSymbol) const

Calls the function with the given symbolic name (which must be mangled for C++ function that don't have "C" linkage). The function called should be a non-member function that takes no arguments and returns void; however call() doesn't check that this is so.



status lookup(const string &symbol, void *&value) const

Places in *value* the value of a global symbol exported either from the core or from a module. If there is no such symbol lookup() returns the status code SYMBOL_NOT_FOUND, otherwise it returns OK.

status drop(const vector<shared_ptr<ELF> > &mods)

Removes from the set of known modules the set given by the union of

- The set X of droppable modules whose addresses are given to drop().
- The set depends(X), all droppable modules in the transitive closure of the set of all modules that depend directly on any module in X.

If any module that would be left undropped depends on any module that would be dropped, drop() returns
EVIL_DROP and no modules are dropped.

Each module dropped has its finalization code run, in the order opposite that used for initialization.

If one of the addresses given is not the address of a known module then drop() returns MODULE_NOT_FOUND without dropping any modules.

status Linker::drop()

Like drop() with an argument but attempts to drop all modules flagged as droppable.

status clear()

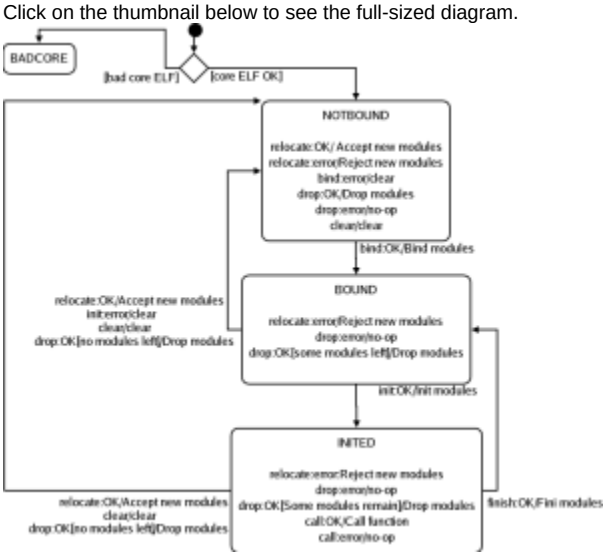
Like drop() but drops all modules including those flagged as undroppable. Always returns OK and returns the linker to its initial state.

linkerstate Linker::state() const

Returns the current state of the linker state machine.

Linker state transitions

Click on the thumbnail below to see the full-sized diagram.



Only a member function appearing in one of the tables below can change the linker state. If a function is not listed for a particular state it means that in that state it returns OK but does nothing.

Sometimes the new state will depend on the status code returned, in which case the relevant codes are shown in the Operation column. If a particular combination of operation and return code doesn't appear for a state, either the combination is impossible for that state or has no effect whatever.

Passing an empty list of modules to relocate or drop operations results in no change of state.

The initial state is NOTBOUND unless the core dynamic section passed to the constructor is found to be badly formed, in which case the initial state is BADCORE.

BADCORE

In this state all operations do nothing except return BAD_ELF_OBJECT; the linker is useless.

ERROR

Beahve much like BADCORE except that operations return INTERNAL_ERROR. This state is entered when an operation returns a status of INTERNAL_ERROR.

NOTBOUND

Meaning: Some modules may have been added using relocate() but not all have been bound. This is the only non-error state in which there may be no known modules.

Operation	Return code	New state	Action
relocate	OK	NOTBOUND	Relocate new modules, add to known set.
relocate	BAD_ELF_OBJECT	NOTBOUND	Reject all new modules.
relocate	DUPLICATE_MODNAME	NOTBOUND	Reject all new modules.
relocate	DUPLICATE_DEFINITIONS	NOTBOUND	Reject all new modules.
bind	OK	BOUND	All references to the core and between modules have been resolved.
bind	UNDEFINED_REFERENCES	NOTBOUND	As many references as possible have been resolved.
bind	BAD_ELF_OBJECT	NOTBOUND	None.

drop	OK	NOTBOUND	The requested modules and those that depend on them have been dropped.
drop	MODULE_NOT_FOUND	NOTBOUND	Nothing gets dropped.
drop	EVIL_DROP	NOTBOUND	Nothing gets dropped.
clear	any	NOTBOUND	All modules get dropped, even "undroppables".
init	TOO_SOON	NOTBOUND	None.
call	TOO_SOON	NOTBOUND	None.

BOUND

Meaning: There are known modules and all have been bound; no unsatisfied references remain.

Operation	Return code	New state	Action
relocate	OK	NOTBOUND	The new modules are relocated and added to the known set.
relocate	BAD_ELF_OBJECT	BOUND	Reject all new modules.
relocate	DUPLICATE_MODNAME	BOUND	Reject all new modules.
relocate	DUPLICATE_DEFINITIONS	BOUND	Reject all new modules.
init	OK	INITED	Initialize all known modules.
init	DEPENDENCY_CYCLES	NOTBOUND	None.
init	INIT_ERROR	NOTBOUND	None.
drop	OK	(no. of modules) > 0 ? BOUND : NOTBOUND	The requested modules and those that depend on them have been dropped.
drop	MODULE_NOT_FOUND	BOUND	Nothing gets dropped.
drop	EVIL_DROP	BOUND	Nothing gets dropped.
clear	any	NOTBOUND	All modules get dropped, even "undroppables".
call	TOO_SOON	BOUND	None.

INITED

Meaning: There are known modules all of which have been bound and initialized.

Operation	Return code	New State	Action
relocate	OK	NOTBOUND	The new modules are relocated and added to the known set.
relocate	BAD_ELF_OBJECT	INITED	Reject all new modules.
relocate	DUPLICATE_MODNAME	INITED	Reject all new modules.
relocate	DUPLICATE_DEFINITIONS	INITED	Reject all new modules.
drop	OK	(no. of modules) > 0 ? INITED : NOTBOUND	The requested modules and those that depend on them have been dropped.
drop	MODULE_NOT_FOUND	INITED	Nothing gets dropped.
drop	EVIL_DROP	INITED	Nothing gets dropped.
finish	OK	BOUND	Finalization has been performed for all known modules.
call	OK	INITED	The requested module function has been called.
call	SYMBOL_NOT_FOUND	INITED	No module function has been called.
clear	any	NOTBOUND	All modules get dropped, even "undroppables".