Building shared objects for ARM RTEMS

Types of shared object

In our refactoring of RTEMS we've created several different types of shared objects. All are ELF dynamic objects in overall structure but each plays a different role:

- RTEMS object (rtems.so). This contains most of RTEMS, newlib and other basic runtime support. It's unique in that it's not loaded by the dynamic linker; instead it's loaded by U-Boot at a fixed location, the start of the Run-Time Support Region of memory. U-Boot loads each segment of rtems. so at the so-called physical address of the segment so we have to use a special linker script to set those properly. The RTEMS object is permanently resident, a.k.a. installed.
- Symbol-Value Tables (*.svt).The dynamic symbol table of this type of object is used as a read-only lookup table whose content is defined by C source code. Each symbol's value is the location of a data object, generally a string or a struct. There can be up to 32 SVTs numbered 0-31, with number 31, the System table sys.svt, and number 30, the Application table app.svt, being loaded at system startup.
- Ordinary shared objects, a.k.a. shareables (*.so). Basically add-on libraries that are installed when needed, e.g., nfs.so and console.so.
- Tasks (*.exe). A task is roughly equivalent to a main program. The loader and dynamic linker cooperate to load a task and, when necessary, to load and install any shareables it needs. A task also has an associated thread which executes the task code starting at its entry point, always present and always named Task_Entry(). The properties of the task such as name, priority, etc., are usually looked up in an SVT. One or more tasks are automatically chosen, using more information stored in SVTs, to be loaded and run at system startup.
- Device drivers (*.drv). These objects perform device initialization and may register RTEMS device drivers.

All types of objects except rtems.so:

- Are loaded into storage that is allocated dynamically from the RTS Region.
- May have a function called lnk_prelude() which is called after the object is fully linked and initialized but before the entry point, if any, is called. For instance driver objects have no entry point but do their work in their lnk_prelude() functions.
- May contain a 32-bit word named lnk_options which is a bit-mask of dynamic linker options defined in tool/elf/linker.h:
    - LNK_INSTALL. Install this shared object.
    - LNK_USE_PREFERENCES. Find a preferences structure for the object and pass its address to lnk_prelude().

The lnk_prelude() function

Consider this the last stage of the construction of the object's image in memory. The image should not be considered complete before this function is run, so it's a programming error to have the image used by other images before lnk_prelude() has finished running.

lnk_prelude() takes two arguments and returns a standard facility/error code. The arguments are two void-pointers:

1. prefs. Either NULL or the address of a preferences structure for the image. The function must cast the pointer to the type that's correct for the image.
2. elfAddr. The address of the ELF file header.

Loadable segments

A shared object may have any number of loadable segments. One of them must contain the ELF file header and the program headers. Since the file header starts at the offset zero in the file then the segment that contains it will also have an file offset of zero and will usually have the lowest address as well. After loading the access permissions of the memory containing a segment are changed to match the permissions recorded in the segment. Permissions can change only at 4K boundaries in the RTS region so segments must be aligned tho 4K boundaries. We recommend the following set of segments:

1. Read-only data containing the file header and the program header table.
2. Executable, read-only text.
3. Read-write data.

Versions of ld available

Target platform	*ld* version	Notes
Native Intel RHEL5	2.17	Doesn't support -l:*filename*
Native Intel RHEL6	2.20	
ARM RTEMS 4.11	2.23+	
ARM XIlinx Linux GNU EABI	2.23+	
ARM Xilinx EABI	2.23+	

We need the syntax -l:*filename* if we find shared libraries using -L and want to use arbitrary file names for them. I list the Intel linkers here in case someone wants to try, for example, a quick test of new linker script techniques without setting up the cross-compilation environment.

Required command-line options

Driver control

Option	Notes
-B ${RTEMS_ROOT}/tgt/arm-rtems4.11/*bsp-variant*/lib -specs bsp_specs -qrtems	Compile and/or assemble and/or link for RTEMS.
-nostdlib	Don't make the linker search standard language or system libraries.
-nostartfiles	Don't tell the linker to link in the standard shared-object startup files.
-o *output-file*	Put the output in the given file.
-c (optional)	Compile and assemble only, producing relocatable object code.
-S (optional)	Compile only, producing assembler source code.
-E (optional)	Preprocess only, producing pre-processed source code.

C/C++ compilation

Option	Notes
-fPIC	Shared objects for ARM must use position-independent code. *ld* will check to make sure you've used it.
-Wno-psabi	Prevents warnings about the implementation of stdarg.
-Wall	Enables all other warnings.
-march=armv7-a -mtune=cortex-a9	Produce assembly code for the ARM Cortex-A9.
-DEXPORT='__attribute__((visibility ("default")))'	Use this macro in declarations in order to put the corresponding symbols into the dynamic symbol table.

Assembly code

Option	Notes
-x assembler-with-cpp	Allow preprocessor directives in the input source code.
-P	Omit #line directives in the preprocessor output, the assembler doesn't accept them.

Static linker

Some of these options need to be prefaced with "-Wl,". However a series of options can be given following a single preface, e.g., "-Wl,-soname=foo,--hash-style=gnu,-zcombreloc".

Option	Needs -Wl,	Notes
-shared		To make a shared object.
-e or --entry		Specifies the entry point (use 0 if there is none).
-soname	✅	Make sure SO names are used in *needed* lists.
-zcombreloc	✅	Combine relocation entries into one or two tables and sort them by the index of the symbol referred to. This allows symbol lookups to be cached by the dynamic linker for efficiency.
--zmax-page-size=4096	✅	This is the page size used in the Run Time Support Region, the part of memory into which shared objects are loaded. Controls the alignment of segments.
--hash-style=gnu	✅	Provides more efficient symbol lookup.
-l:*libname*		Refer to a shared library *libname* that's to be found using the search path established using -L options.
-L *dirname*		Add *dirname* to the search path for *ld*, which uses the path for two purposes: to find libraries named using -l: and to find linker scripts named in *include* statements in other linker scripts (and named using -T).
-T *scriptname*	✅	Use *scriptname* as the main linker script.
--no-undefined	✅	Makes it an error if the resulting shared object has undefined references remaining after *ld* has finished making it. It allows needed objects to contain undefined references but presumably these too have been constructed using --no-undefined.

File names, sonames and the *needed* list

A shared library has both a file name and a so-called SO (shared object) name, or soname. The two need not be related as the soname can be set using the *ld* option -soname. The soname is contained in the dynamic string table at an offset given by the DT_SONAME entry in the dynamic section. Other entries of type DT_NEEDED, which also contain offsets into the dynamic string table, specify the *needed* list for the shared library. These are the shared libraries that must also be loaded for the shared library to work. Standard convention allows the needed list to contain both file names and sonames but we'll use only sonames.

File name conventions

File names need not begin with "lib"; we use -l: or just name the shared object file as input. We use the following extensions for the different kinds of shared objects:

- **.svt** for symbol-value tables.
- **.exe** for tasks.
- **.drv** for device drivers.
- **.so** for everything else.

Soname and *needed* list conventions

1. Every shared object must have a soname.
2. Sonames must be legal C identifiers.
3. Sonames must be no longer than 128 characters.
4. The *needed* list for an shared object must contain a reference to each other object on which it depends.
5. A shared object must have no undefined references left after linking with *ld*.
6. The following sonames are reserved:
    a. "RTEMS___" for the shared object containing RTEMS, newlib and other run-time support.
    b. "SYS" for the shared object containing the System Name Table.
    c. "APP" for the shared object containing the Application Name Table.

In general a shared object is found by using Svt_Translate() on the soname in order to obtain the full path name, where the soname is gotten from a *needed* list. The Symbol-Value tables used by Svt_Translate() are created by compiling C code which is why the sonames must be legal C identifiers. The length restriction on sonames comes from the RTEMS dynamic linker which for efficiency's sake uses a fixed-size buffer to create SVT keys from sonames. Sonames are also the keys used for the database of installed objects.

In order to have a shared object satisfy the requirements listed above we need to use -soname when building every shared object. Then whether you include the object as an input or search for it using -L and -l:, *ld* will put the soname on the *needed* list. Referencing an object that lacks an embedded soname will result in the file name of the object being put on the *needed* list, and that will cause the lookup with Svt_Translate() to fail.

All the direct dependencies of the shared object being built should be searched or included as inputs, and undefined references should cause the build to fail (--no-undefined). Almost always you should name only the objects to which the one being built makes symbolic references, as ld by default puts every object so named on the *needed* list. If you have to you can get ld to filter out those objects that don't satisfy symbolic references by using the option --as-needed early in the command line. You can turn this mode off in the rare cases in which you really need an object that doesn't satisfy some symbolic reference: use --no-as-needed. If you need to build an object a.so that might make symbolic references to b.so, c.so or d.so, and which doesn't make such references to e.so but still needs it, you command line would look something like this:

```
arm-rtemsx.yy-g++ -shared -o a.so -soname A ... a.o  --as-needed b.so c.so d.so --no-as-needed e.so
```

Dynamic symbol table

1. Every shared object must have a GNU-style hash table (--hash-style=gnu). System V hash tables will be ignored and should not be generated.
2. Certain symbols are used for data and functions used by the dynamic linker. Currently these are:
    a. "lnk_preferences" which labels object preference data to be passed to lnk_prelude().
    b. "lnk_prelude" which labels a function to be called by the dynamic linker just after the functions pointed to by the .init_array have been run.

The special symbols must have global visibility in order for the dynamic linker to see them. However, it treats them as strictly local definitions and won't make cross-object references with them.

The ARM cross-compilers use .init_array to hold pointers to compiler-generated functions that run constructors for statically allocated C++ objects. Therefore .init_array is processed before calling lnk_prelude(). However, lnk_prelude() can't itself be called using the .init_array mechanism because it takes arguments, it returns a status value and there's no way to control the ordering of entries in .init_array.