Chapter 10 Creating and maintaining a library

This chapter describes how to make and maintain a library using the program named libr86. After that, the procedures how to compile libraries of the LSI C86 are described.

10.1 libr86

The libr86 is a kind of the compiler driver that disassembles the library modules, which are gathered into one file, compiles them, and combines them with library files. This section describes how to operate the libr86.

10.1.1 Library file

Generally, a library file is created by compiling multiple modules to make object files and combining these object files by the librarian. To make disused modules not included in the execution file, it is necessary to separate these modules into as many modules as possible. That is, it is ideal that one module contains only one function or subroutine.
To achieve this, each library function is put in different file and all files must be compiled.

However, the libr86 does not use such way for reason listed below.

- Functions, which have strong relationship, must be put in the same file.
- As the number of files becomes bigger, editing of them may become difficult.

Instead of the above way, multiple modules are put in one file and they are disassembled and converted into the library format only during compiling and assembling.
The libr86 is used to perform this task.

10.1.2 Library source codes

Library source codes processed by the libr86 has the following format.

```
- -
     :
     common header 1
     :
-module 1
     :
     main body of module 1
     :
-module 2
     :
     main body of module 2
     :
- -
     :
     common header 2
     :
-module 3
     :
     main body of module 3
     :
-module 4
     :
     main body of module 4
     :
```

A line starting with - (hyphen) is a command to the libr86. Therefore, no hyphens are put at the beginning of a line in the program main body.
A line starting with two hyphens shows that a common header is put in the subsequent lines. The common header is added to the beginning of the subsequent modules. Definitions commonly used by all modules are put in the common header. If a line starting with - - appears again, the previous common header becomes invalid. The common header and module main body are put any times.

A line starting with one hyphen (-) shows that subsequent lines are handled as one module. A name put immediately after the hyphen (-) is handled as a file name of the module. The libr86 creates a new file with this name, copies the common header and main body into this file, and calls up the lcc86 to start the compilation.
When the object files are made, these files are combined with the library file named "`libr.out`" and the previously created file is deleted.

The module can be written in either C or assembly language. Both the libr86 and lcc86 judge the programming language based on the file name put after the hyphen (-). Therefore, the specific extension, such as .c for the C language, .a86 for the assembly language, or .p86 (extension for assembling with `ccp put`), which the lcc86 can clearly understand, must be put.

The following shows an example.

```
/*
*      sample.l - sample library file
*/
- - C program part
#include <ctype.h>
#include <string.h>

-stolower.c
char   *stolower (char *s)
{
       char   *p;
       for (p = s; *p; p++)
              *p = tolower (*p);
       return (s);
}
-stoupper.c
char   *stoupper (char *s)
{
       char   *p;
```

```
        for (p = s; *p; p++)
                *p = toupper (*p);
        return (s);
}

- - Assembly language part
CGROUP GROUP  TEXT
TEXT   CSEG

-ror.a86
ror_::
        push   cx
        mov    cl, bl
        ror    ax, cl
        pop    cx
        ret

-rol.a86
rol_::
        push   cx
        mov    cl, bl
        rol    ax, cl
        pop    cx
        ret
```

In the above example, a library having four modules is defined, two modules are written in the C language, and other two modules are written in the assembly language.

When this library is processed by the libr86, four modules, `stolower.c`, `stoupper.c`, `ror.a86`, and `rol.a86` are generated sequentially, and then they are compiled.

Two #include statements are put at the beginning of the modules `stolower.c` and `stoupper.c`. Pseudo commands `GROUP` and `CSEG` are put at the beginning of the modules, `ror.a86` and `rol.a86`.

10.1.3 Operating the libr86

A libr86 command has the following format.

```
libr86 [option] [lcc86 option] input file name
```

The input file name specifies a name of the file containing library source codes. Any file name can be used, but normally an extension of `.l` is put.
The libr86 reads the input file, generates modules one-by-one, compiles them, and combines them with the library file.

In the actual compilation, the lcc86 is called up. Modules are combined by the librarian `oar`. Therefore, these commands must be placed in an executable directory.

A work file is made in the current directory. A library file named `libr.out` is made in the current directory.
The following options are also provided.

`-l`	Microsoft's LIB is used as librarian instead of `oar`.
`-a`	Compiling and making of library are not executed. Only commands are shown.
`-o file`	A library file name to be output is set to file. Unless this option is specified, the output file name becomes `libr.out`.
`-s`	Normally, the libr86 displays lcc86 or oar commands running internally on the console. If this option is used, they are not displayed.
`-x lcc`	`lcc` is used as compiler driver. If this option is omitted, the lcc86 is used as compiler driver.

All other options starting with a hyphen (-) are transferred to the lcc86.
For example, to generate the library file `sample.lib` from the `sample.l` in the previously described list, the following command is executed.

```
libr86 -ms sample.l
```

Where, `-ms` is an option of the lcc86 that specifies a small model. As a result, a library file named `libr.out` is made in the current directory. When this file is renamed to `sample.lib`, operation of the libr86 is completed. (In the UNIX environment, the `mv` command is used instead of the `ren` command.)

```
ren libr.out  sample.lib
```

10.2 Creating a LSI C-86 library

This section describes how to make a LSI C-86 library file.

10.2.1 Library source codes

Library source codes of the LSI C are stored in the directory "`instpath\src\lib`" made by the installer. If the library source codes are not installed, they must be installed before creating a library.
The above directory has a structure shown in Fig. 10.1 Among these files, the `crom.p86` is used to make a ROM program.

10.2.2 Creating a library

The following shows how to compile a library.

1. First, make a directory containing necessary files. Copy library source codes into an appropriate directory.
   If your system contains only two floppy disk drives (probably MS-DOS environment), put files listed in Fig. 10.2 in floppy disks A and B. After that, set the environmental variable `PATH` to make execution of the `lcc86` and `libr86` possible.

   ```
   set PATH = A:\; A:\bin
   ```

2. Next, change the `makefile` depending on the environment. Rewrite three macros, `LSIC_PLATFORM`, `TARGET`, and `INTER` at the beginning of the `makefile`. In `LSIC_PLATFORM`, specify `dos` for MS-DOS, `w32` for Windows, and `unix` for UNIX, respectively. In `TARGET`, specify a directory containing created library files. In `INTER`, specify a directory containing intermediate files made by the `libr86`. However, the memory model designation made finally must be omitted in these directories.

   For example, to compile the library source codes using two floppy disk drives, specify the following.

   ```
   LSIC_PLATFORM = dos
   TARGET = B:\lib
   INTER = B:\src\lib
   ```

3. Finally, create directories which are specified in `TARGET` and `INTER`.

Let's make a library. First, change the current directory to that containing source files, and then type the following command.

makefile	makefile for creating a library
makefile.dx	Subcommand of makefile
makeone	Subcommand of makefile.dx
conf.*	Files that the makefile includes.
cdos.p86	MS-DOS initialization program
shell.l	Upper level main function
dos.l	MS-DOS interface function
error.l	Error processing function
farmem.l	far memory management function
farstr.l	far character string processing function
float.l	Floating-point calculation package
fmtio.l	Formatted input/output function
general.l	General function
io.l	High level input/output function
kanji.l	Japanese character (string) processing function
known.l	Run-time routine called by subroutine
math.l	Mathematics function
memory.l	Memory allocation function
signal.l	Signal function
string.l	Character string processing function
time.l	Time function
expand.c	Function that makes wildcard development matched with MS-DOS
nonexpand.c	Function that cancels wildcard development
tinymain.c	Function that cancels parameter quart and wildcard development
crom.p86	Initialization routine for ROM program

Fig. 10.1 Library sources

```
A:\                         System disk
       MSDOS.SYS
       IO.SYS
       COMMAND.COM
       EDLIN.EXE            Or other editor

       bin\                 Execution files
              kmmake.exe
              makedef
              libr86.exe
              oar.exe
              lcc86.exe
              _lcc86
              cpp.exe
              cf.exe
              cg86.exe
              r86.exe

       include\
              *.h
              sys\
                     *.h

B:\                         Work disk
       src\
              lib\   Copy of contents in instpath\src\lib\
              makefile
              makefile.dx
              makeone
              conf.dos
              *.l
              *.c
              *.p86
              s\            Directory for intermediate results

       lib\
              s\            Directory for created libraries
```

Fig. 10.2 Disks for creating libraries

```
kmmake s
```

This makes an S model library in the directory which is defined in TARGET. Accordingly, specifying P, D, and L for target names will make libraries for relevant models. If “all” is specified for the target name, libraries for all models are made at once. (When you compiles using only two floppy disk drives, it is recommended to make a library one-by-one while carefully observing the disk capacity.)