Chapter 12 Linker lld

This chapter describes the linker lld.

12.1 Overview

The lld is a linker that is compatible with Microsoft's LINK. The lld reads object files and library files, and combines them to create `*.exe` and `*.hex` files.
Note that the lld does not support the debug information records for CodeView.

12.2 Operating the lld

The lld is called up in the following format.

```
lld [options] objectfiles…
```

In above *`objectfiles`*, names of object files and library files to be input are specified.
The following options are also provided.

`@`*`file`*	Command line parameters are read from file.
`-F`*`form`*	This sets the output file format to "form". This form must be any of the following.

e	MS-EXE format
c	COM format
h	Extended Intel HEX format
s	Motorola S2 format

If no above options are given, the output file format is determined based on its extension.

	`.exe` e
	`.com` c
	`.hex` h
`-I [X]`	This controls whether or not the upper and lower cases are distinct. `-I, -I1` Upper and lower cases are not distinct. `-I0` Upper and lower cases are distinct If above options are not given, the default setting "`-I0`" is applied.
`-Ldir`	This specifies a directory where library files are searched for.
`-lfile`	This scans the library "file". Extension, ".lib", can be omitted.
`-g`	This generates line number information. -M is also given automatically.
`-M`	This generates a map file.
`-o file`	This sets the output file name to "`file`". If this option is omitted, a `.EXE` file with the same base name as that of the file which appears first in the objectfiles.
`-s hex`	This sets the size of the stack to "`hex`". hex is expressed in hexadecimal notation and has a 16-bit value.
`-T hex`	This sets the start address of the first segment to "`hex`". hex is expressed in hexadecimal notation and has a 20-bit value.
`-Tseg hex`	This sets the start address of the segment "`seg`" to hex. hex is expressed in hexadecimal notation and has a 20-bit value.

12.3 Segment placement

The `lld` reads object files in order specified by the command line to combine them. At this time, each segment is placed according to the following rules.

1. Each segment is arranged in order of that segment pseudo command appears on the source program.

2. Segments with the same name and a connection type of `PUBLIC`, `MEMORY`, `STACK` or `COMMON` are gathered. Segments with a connection type of `PUBLIC` or `MEMORY` are simply overlapped in input order. The total size is a sum of each segment size input.

Segments with a connection type of `COMMON` are overlapped. The total size becomes the same as a maximum size among input segments. After that, the segments are overlapped so that the beginning of each segment is aligned.
Segments with a connection type of `STACK` are also overlapped. However, they are overlapped so that the end of each segment is aligned, different from `COMMON`. The total size is a sum of each segment size input.

3. Segments belonging to the same class are placed continuously.

4. An address is allocated to each segment in order. If `-T` option is not specified, the start address becomes `0` and an address satisfying the alignment attribute is allocated sequentially. If a segment for which an address is specified by `-T` option is encountered, addresses are allocated from that address in order.

For example, assuming that the following segments are aligned in a program.

```
TEXT          CSEG   ‘CODE’
DATA          DSEG   ‘DATA’
XSTACK DSEG   ‘STACK’
BSS           DSEG   ‘DATA’
```

The linker places segments in order of that they appear in object files. At this time, segments with the same class name are gathered. Therefore, programs are arranged as shown in Fig. 12.1. With default settings, the lld links the segments as the address of the first segment is 0.

TEXT	‘CODE’
DATA	‘DATA’
BSS	‘DATA’
XSTACK	‘STACK’

Fig. 12.1 Segment arrangements

Use of `-T` option provided by the `lld` makes it possible to specify an absolute address of the segment. There are two ways how to use `-T` option.
When using `-T` option without specifying of a segment name, the segment that appears first is relocated at a specified address. Since subsequent addresses are then placed sequentially from that address, the segment placement becomes the same as that without use of `-T`. Only the first segment becomes different.
At this time, note that 5-digit hexadecimal number must be used to specify an address. For example, when `-T 20000` is specified, the first segment is placed at `2000:0000`.

When a segment name is specified as -TDATA 12340, the start address of that segment (DATA in this example) becomes 1234:0000. Subsequent segments are placed after that segment in the same manner as described above.

Only the segment name can be specified using the -T option. Group and class names cannot be specified.

12.4 Ghost segment

When a program is stored into ROM, it may be required to store data to a place at an address different from actual one. The lld provides the ghost segment function to achieve this requirement.
The ghost segment is a segment that begins with a name of _GHOST_. If the lld finds a segment with a name of _GHOST_name in the object module, it stores the entity of the segment with "name" at that address.

For example, let's see the following program.

```
TEXT   CSEG
       mov    ax, DATA
       mov    ds, ax
       mov    ax, [my_data]

_GHOST_DATA  CSEG   PARA

DATA   DSEG
dummy: dw     1234h
my_data::     dw     1
```

The file name, "test.a86", is put on the above program. When this program is assembled and linked using

```
lcc86 -a -g -k"-Fh -o test.hex -T 10000 -TDATA 20000" test.a86
```

a HEX file having the contents shown in Fig. 12.2 is created. (In the UNIX environment, it is recommended to enclose using ' instead of ".)
As shown in the Fig., the contents of the DATA segment are moved to _GHOST_DAT, and then output. The DATA segment itself is not output.
However, labels in the DATA segment are correctly referred to from the start offset of the DATA segment.

If the contents of this ghost segment are moved to actual addresses using the initial routine during execution, it becomes possible that the initialization data is placed in the ROM and RAM addresses can be referred to from the program.
For details about storing of programs into ROM, see Chapter 13, "ROM programming".

```
:020000021000EC
:08000000B800208ED8A1020017
:020000021001EB
:0400000034120100B5
:00000001FF
```

Address	Command	Segment
10000	`mov  ax, 2000h      ;DATA`	TEXT
10003	`mov  ds, ax`	
10004	`mov  ax, [0002h]    ; my_data`	
10010	`dw  1234h`	_GHOST_DATA
10012	`dw  1`	

Fig. 12.2: Created HEX file

12.5 Library searching sequence

The `lld` adopts a library search method different from Microsoft's LINK. Even though modules in each library file refer to each others, the LINK solves it. The `lld` searches each library file only once. Therefore, if modules in each library file refer to each others, the symbol becomes undefined in the `llb`.

The `lld` processes given options in order. If the `lld` finds `-l` option, it searches the library for unsolved symbol. It seems inconvenient, but use of this feature makes it possible to do the following.

```
  lld -o  foo.com  foor.obj  -llibc  foonr.obj  -llibc
```

Assuming that `foo.com` is TSR (resident program) and that `foor.obj` contains resident codes of `foo.com` and `foonr.obj` contains non-resident codes.
`libc.lib` is a library. When the above command is executed, the first `-llibc` combines only library routines which are referred from the resident part. Then, the last `-llibc` searches for non-connected library routines of those referred from the non-resident part.
That is, it is possible to split the resident part and non-resident part, and to use library routines.