UniFLEX™ Linkage Editor

MANUAL REVISION HISTORY

Revision	Date	Change
A	9/81	Original Release
B	12/81	Addition of Chapter 8

Table of Contents

1. INTRODUCTION

1.1. Preface

This manual describes the use and operation of the UniFLEX Linking Loader. It is assumed the reader is familiar with the operation of the UniFLEX 6809 Relocating Assembler, and is comfortable with the concepts of relocation and linkage editing.

Throughout the manual a couple of notational conventions are used which are explained here. Angle brackets ('<' and '>') are often used to enclose the description of a single item even though the description may require several words. Square brackets ('[' and ']') are used to enclose an optional item.

1.2. Terminology

file
    A UniFLEX file containing one or more relocatable object code modules.

file name
    The name of a UniFLEX file.

loading
    The placement of instructions and data into memory in preparation for execution. This preparation includes linking (the matching of symbolic references and definitions), and relocation of symbols and address expressions.

module
    A subprogram which has been assembled using the 6809 relocating assembler.

module name
    The name given to a module by the programmer by using the "name" directive of the relocating assembler. If the "name" directive was not used, the module name is the same as the UniFLEX file name in which it is contained. Therefore, several modules may have the same name. The output module of the loader may be given a name by use of the "N" option.

relocatable object-code module
    Equivalent to "module".

1.3. Linking Loader Input

Technical System Consultants ~~Inc.'s Linking~~ Loader will accept as input independently assembled, relocatable object-code modules. Relocatable object-code is generated by the UniFLEX relocating assembler "relasmb" in such a way that addresses are not bound to absolute locations at assembly time; this binding of the address fields will be accomplished by the Linking Loader. "Link-edit" binds the addresses at the time the object-code segments are combined to produce an executable program. The binding or adjustment of the address fields is termed "relocation". Relocation is necessary when an instruction expects an absolute address as an operand. The address field of this instruction must be increased by a "relocation constant". The relocation constant is the address at which the module is loaded for execution.

Address fields which do not require relocation are absolute addresses (their values remain the same regardless of the position of the object-code segment in memory). Since the loader does not have access to the source text, it cannot determine if an address field is absolute or relocatable. In fact, it cannot distinquish addresses from data or opcodes. Therefore, the assembler must indicate to the loader the address fields which require relocation. This communication is accomplished through "relocation records" which are appended to the object-code file produced by the assembler. Such a file is called a "relocatable object-code module".

Often it is desirable for parts of a program (called modules) to be developed separately. Each module must be assembled separately prior to final merging of all the modules. During this merging process of the modules, it is necessary to resolve references in a module which refer to addresses or data defined in another. The resolution of "external references" is called linking. The assembler must provide information to the loader, in a manner similar to relocation records, concerning the address fields which must be resolved.

1.4. Linking Loader Output

As output, "link-edit" produces an object-code module, a load map, a module map, and a global symbol table. The object-code module can be either relocatable or executable. A relocatable module produced by the loader cannot be distinquished from a relocatable module produced by "relasmb". Only the loader, however, can transform multiple relocatable modules into an executable program.

2. INVOKING THE LOADER

The Linking Loader accepts as input previously assembled, relocatable object-code modules and produces as output:

1) A link edited, relocatable, object-code module or
2) a link edited, relocated, executable program.

The command line necessary to invoke the linking loader is as follows:

```
++ link-edit <relocatable modules> [+options]
```

where:

the two plus signs are UniFLEX's ready prompt, and "link-edit" is the name of the linking loader command file.

"relocatable modules" is a list of one or more UniFLEX file names, separated by blanks, of the relocatable object-code modules you wish to load. The object-code modules will be loaded in the order specified.

"options" is a list of options which must start with a plus sign ("+") and may not contain any embedded spaces. More than one list of options may be specified, but each list must start with a plus sign. Some of the options are single characters while others require an argument. Those that are single letters may be grouped together; for example: +up. Those that require arguments may either stand alone or be the last of a group of options; for example: +pma=100, where the "a=100" is an option with an argument. The equal sign is not required in options with an argument. Therefore, "+a=100" is equivalent to "+a100".

Following is a detailed description of each of the valid options.

+a=addr
The binary output of the linking loader is to be an absolute, executable program, and its beginning load address (in hex) is to be "addr". The effect of this option is as if all the modules were contained in one source file and assembled with the absolute assembler using an origin of "addr". See "Absolute Programs" in the SEGMENTATION chapter for a more complete description.

+g=<go time command line>
Specifies that after the program has been link-edited and relocated, execution control is to be passed to the program, starting at the address specified by the transfer address. Since the "g" option signifies the start of the arguments to be passed to the program being linked, the "g" option must be

the last option specified on the command line. Perhaps an example will make this more clear. Suppose the UniFLEX utility "echo" is composed of three modules: "echo1.r", "echo2.r", and "echo3.r". We wish to link these three modules to produce the program "echo" and immediately begin execution of "echo" passing it the parameters "+7" and "Hello there". The command line to accomplish this would appear as follows:

```
     ++ link-edit echo[1-3].r +no=echo +g=+7 "Hello there"
```

Option "a", "n", or "t" must be specified when using "g". The linking loader executes the resultant program directly. Therefore, no shell features such as file name matching can be used.

+i Include all internal symbols in the symbol table for symbolic debugging. If the "i" option is not specified, only global symbols are included in the relocatable, object-code module.

+L Do not search the libraries for unresolved externals.

+l=<library file name>
A maximum of five libraries may be specified by repeated use of the "l" option. If less than five libraries are specified, the system library "Syslib" is also searched in addition to the user libraries. Libraries are searched only when an executable output program is specified (by the use of the "a", "n", or "t" option). In the following example, an effort is made to resolve externals not found in the user's modules by searching the three libraries libl, lib2, and Syslib.

```
     ++ link-edit echo[1-3].r +n +l=libl +l=lib2
```

See the LIBRARIES chapter for more information concerning the formation and use of libraries.

+m Print the Load Map and the Module Map. The Load Map provides information as to the type of output file produced, the length of the resulting output object-code module, the number of input modules, and the transfer address. The Module Map describes the load address and object-code length for each of the input modules.

+N=<module name>
The name "module name" is to be given to the output module of the loader in a manner similar to the "name" directive of the relocating assembler. Since the loader does not propogate the module names of the relocatable input modules to the output module, the "N" option must be used to assign a name to a module. If the "N" option is not used, the module name will default to the name of the file in which it is contained. Both executable programs and relocatable modules can receive module names. The name is limited to a maximum of 14 characters.

+n The binary output of the linking loader is to be a no-text, executable program. The load address for all no-text programs is 0. The effect of this option is as if all the modules were contained in one source file and written in such a manner that the size of the text segment is 0. See "No-Text Programs" in the SEGMENTATION chapter for a more complete description.

+o=<file name>
Allows specification of the output binary file name. If the "o" option is not specified, the output file name will be "output" in the current directory. If a file by this name already exists, it will be deleted.

+p Selects pagination of the printed output. The date (if the "d" option is not specified) and a page number are included at the top of each page.

+r The binary output of the linking loader is to be a relocatable object-code module. If neither the "a", "n", nor "t" options are specified, "r" is assumed. The effect of this option is as if all the modules were contained in one source file and assembled with the relocating assembler.

+s Select printing of the Global Symbol Table. If specified, the linking loader will print each global symbol and its address.

+t The binary output of the linking loader is to be a shared text, executable program. The load address for all shared text programs is 0 for the text segment, and the next 4K boundary after text is the beginning of the data segment. The uninitialized data segment immediately follows the data segment. The effect of this option is as if all the modules were contained in one source file and written in such a manner that a shared text binary file was produced. See "Shared Text Programs" in the SEGMENTATION chapter for a more complete description.

+u Do not print the "unresolved external" message when producing a relocatable output module.

3. LIBRARIES

3.1. Introduction

The linking loader has the capability of searching a maximum of five libraries when there remain externals which cannot be resolved from the user's modules. Usually, these libraries consist of up to four user libraries and the system library, "/lib/Syslib". The user can, however, specify five libraries on the command line. When five libraries are specified, the fifth one takes the place of the system library.

When searching for a library, the linking loader first looks for the specified file in the current directory. If not found, it then looks for the directory "lib" in the current directory. If found, the linking loader attempts to find the specified file in that "lib" directory. If not found there, the loader makes a third and final attempt to find the specified file by looking in the directory "/lib". If not found in any of these three directories, an error message is issued and the loader aborts. This process also is followed when searching for the system library "Syslib".

A library is a special collection of relocatable modules. When an external cannot be resolved from the user's modules, the libraries are searched in an effort to resolve it. The linking loader will search the user defined libraries in the order specified on the command line before searching the system library. This allows the user to redefine a system library module or entry point. The search for an external can be summarized as follows:

1) Can the external be resolved from the user's modules?
2) Can it be found in the user specified libraries?
3) Can the external be resolved from the system library?

When an external is resolved from a module contained in a library, that module is loaded and is then considered to be a "user" module. Because of this, library modules can reference other library modules.

3.2. Library Generation

The "lib-gen" utility is used to create new libraries and update existing libraries. All modules in a library <u>must</u> have a name. The name is assigned to a module by the "name" pseudo-op in the relocating assembler or by the "N" option of the linking loader. It is the responsibility of the programmer to ensure that all modules in a library have names. The "lib-gen" utility will not accept a module without a name.

The "lib-gen" utility is called with a command of the following general form:

```
lib-gen o=<old>,n=<new>,u=<updates>,<options>,<deletions>
```

The arguments may be specified in any order.

The argument "o=<old>" specifies the name of an existing library file. This library file must have been created previously by "lib-gen". If "lib-gen" is begin called to create a new library (instead of updating an existing one), this argument should be omitted.

The argument "n=<new>" specifies the name of the new library. If this file already exists, it will be deleted before the new library is written. This argument need not be specified when updating an existing library. In this case, "lib-gen" will put the new library in a scratch file, delete the old library file, and rename the scratch file, giving it the name of the old library. It is not permitted to omit both the "o=<old>" and "n=<new>" arguments.

The argument "u=<updates>" specifies the name of a file containing modules that are to be added to the library replacing existing modules in the library if necessary. More than one update file may be specified by repeating the "u=" argument. Up to 9 files may be specified in this way. See the examples below.

As "lib-gen" runs, it produces a report, describing the action that it has taken for each module in the library. The report includes the module name and the file from which it was read (the old library or one of the update files). The options are used to eliminate or shorten this report. If the option "+l" is specified, no report will be produced. If the option "+a" is specified, the report will only contain information about those modules that were replaced, added, or deleted. No information about modules copied from the old library will be given.

The "<deletions>" argument is a list of module names that are to be deleted from the old library. The names may be separated by commas or spaces. If a name is specified that cannot be found in the old library, a warning message is issued. If the "+l" option was specified, no warning is issued.

EXAMPLES

1. Create a new library with the name "binlib" containing modules from the files "one", "two", and "three".

   ```
   lib-gen n=binlib u=one u=two u=three
   ```

   Since a new library is being created, the "o=<old>" argument was omitted. Note that the "u=" argument was repeated for each update file.

2. Update the library named "binlib", adding or replacing records from the file "new". Produce an abbreviated report.

   ```
   lib-gen o=binlib u=new +a
   ```

   Since no new library was specified, the new library will be given the name of the old library.

3. Update the library named "binlib", deleting the modules named "diagonal" and "transpose". Also add new modules from the file "xyz" and write the new library in the file "newlib".

   ```
   lib-gen o=binlib u=xyz n=newlib transpose diagonal
   ```

4. SEGMENTATION AND MEMORY ASSIGNMENT

4.1. Relocatable and Executable Files

In order for the relocating assembler to produce a relocatable output module, the source code must be segmented. In other words, instead of using "org" which would force the relocating assembler to produce executable object-code, the segmentation directives "text", "data", and "bss" are used. The use of "org" or the segmentation directives signals the relocating assembler as to which type of output module to produce: executable or relocatable, respectively. In a similar manner, the linking loader can also produce either an executable program or a relocatable module. Use of the "a", "n", or "t" option causes the linking loader to produce an executable program, and the use of the "r" option will produce a relocatable module. Note that the relocating assembler can only produce one kind of executable program, namely an absolute object-code file. The linking loader, on the other hand, can produce three types of executable files: absolute, shared text, and no-text. The next five sections will discuss how "link-edit" produces the relocatable modules and the three types of executable programs.

4.2. Relocatable Modules

Relocatable modules produced by the relocating assembler have distinct text, data, and bss segments. All of the text object-code appears in the binary file first, followed by all of the data object-code. Since there is no object-code in the bss segment, it is conceptually thought of as following the data segment. The linking loader maintains these distinct segments by combining the text segments of all the relocatable input modules, followed by the concatenation of data segments, and then (conceptually) all the bss segments. In addition, it is guaranteed that the module segments are loaded in the order in which the modules are specified on the command line.

Common blocks (which contain only bss) are not combined with the bss segments of the other modules when producing a relocatable output module. Instead, common blocks retain their identity as separate modules and are appended to the resulting relocatable output module. Common areas will be combined with the bss segments of other modules only when producing an executable program.

Relocatable modules can be given module names by the use of the "name" directive of the relocating assembler. This name is used when printing the module map. If no name was given to a module by use of the "name" directive, the name of the file in which it is contained is printed. When producing a relocatable output module, the linking loader does not propagate any of these module names to the output. To assign an output module a name, use the "N" option when invoking the loader.

Unlike module names, "info" fields are collected from the input modules and carried over to the relocatable output module and ultimately to the executable program.

4.3. Executable Programs

When loading modules to produce an executable program, it is guaranteed that the user modules are loaded first in the order specified on the command line. Common areas (which contain only uninitialized data) are loaded after the last module specified on the command line. Libraries are loaded after the last common block, or after the last user module on the command line if there are no common blocks.

The three types of executable programs which the linking loader is capable of producing have the same form as the same executable program types of the absolute assembler. The three types are absolute, shared text, and no-text.

4.3.1. Absolute Programs

This is the most common type of executable output produced by the linking loader. In this type of file, the binary is stored in a record format where each record has it own load address and contains the object-code of one module. The user can specify the initial program load address (IPLA) of the first record (module). Successive modules are loaded in the order specified on the command line and immediately follow the previous module. In other words, the program is loaded contiguously starting at IPLA. It is highly recommended that all UniFLEX programs load into low memory (usually location 0) to make the most efficient use of the operating system.

Absolute programs do not make the distinction between text (object-code instructions), data, and uninitialized data. However, the modules are assembled using the segmentation directives.

The following memory map illustrates how the individual segments are placed in relation to other segments and modules. The module numbers are the order in which they appear on the command line; "m" is the last module specified. Common blocks 1-x and library modules 1-n which are loaded to complete the program are also represented.

```
                      <--+
IPLA --> Text of mod 1   !   Record 1
         Data of mod 1   !
                      <--+
         Text of mod 2   !   Record 2
         Data of mod 2   !
                      <--+
               .         !   Records 3 to m-1
               .         !
                      <--+
         Text of mod m   !   Record m
         Data of mod m   !
                      <--+
         Text of library 1!  Record m+1
         Data of library 1!
                      <--+
         Text of library 2!  Record m+2
         Data of library 2!
                      <--+
               .         !   Records m+3 to m+n-1
               .         !
                      <--+
         Text of library n!  Record m+n
         Data of library n!
                      <--+
         Bss of  mod 1   !   This memory is allocated to
         Bss of  mod 2   !   the task by UniFLEX at the time
               .         !   execution begins.
               .         !
         Bss of  mod m   !
         Bss of common 1 !
         Bss of common 2 !
               .         !   execution begins.
               .         !
         Bss of common x !
         Bss of library 1 !  This memory is allocated to
         Bss of library 2 !  the task by UniFLEX at the time
               .         !   execution begins.
               .         !
         Bss of library n !
                      <--+
```

4.3.2. Shared Text Programs

Shared text programs have three distinct segments. The "text" segment is assumed to be read-only. This implies the code contained in the text segment will not be altered as long as the program runs. We can take full advantage of this fact by "sharing" this segment among several users who are running the program concurrently. This can mean a considerable increase in the efficiency of the system. Since current versions of UniFLEX perform memory management via 4K pages, the text segment must be contained in its own 4K page(s) in order for multiple users to share the same "program". Therefore, the "data" segment must start on the next even 4K page address beyond the end of the text segment.

The "data" segment is also referred to as "initialized data". It is information (actual instructions or data) which must be initialized or loaded, but which can be altered at some later point. For example, counter variables which must be initialized to zero but will later be incremented should be placed in the data segment. At any time, the variable could be read or its value changed. Each user would then need their own copy of the data segment.

The "bss" segment, like the data segment, is also a read/write area. Since a module does not contain any object-code to be loaded into this section of memory, it is also referred to as "uninitialized data". The module does contain the size of the bss segment, however, in order to inform the operating system that memory is required in this area but does not need to be initialized.

When producing a shared text program, the linking loader collects all the text segments from the relocatable input modules and loads them at location 0. All of the data segments are then placed on the next even 4K boundary after the end of the text segments. Memory for the bss segments will be allocated immediately following the data segments at the time the program is executed.

There are drawbacks to using shared text. The text portion of a shared text file is always swapped to disk. Therefore, programs which are used infrequently, or only one user would be running it at a time, would make better use of the system resources if it were absolute or no-text. Also, if the size of the text segment is small, quite a bit of memory (the remainder of the 4K block) is wasted as evidenced by the memory map below.

The following memory map illustrates how the segments are loaded in relation to other segments and modules. The module numbers are the order in which they appear on the command line; "m" is the last module specifed. Common blocks 1-x and library modules 1-n which are loaded to complete the program are also represented.

```
          0 --> Text of mod 1
                Text of mod 2
                      .
                      .
                Text of mod m
                Text of library 1
                Text of library 2
                      .
                      .
                Text of library n
                              <--+
                                 !  Memory allocated but
                                 !  not used.
                              <--+
     4K page --> Data of mod 1
                 Data of mod 2
                      .
                      .
                 Data of mod m
                 Data of library 1
                 Data of library 2
                      .
                      .
                 Data of library n
                 Bss of  mod 1
                 Bss of  mod 2
                      .
                      .
                 Bss of  mod m
                 Bss of common 1
                 Bss of common 2
                      .
                      .
                 Bss of common x
                 Bss of library 1
                 Bss of library 2
                      .
                      .
                 Bss of library n
```

4.3.3. No-Text Programs

No-text programs have characteristics in common with both shared text and absolute programs. No-text programs are like shared text programs in that both have distinct segments as defined by the use of the segmentation directives of the relocating assembler and maintained by the linking loader. No-text programs, however, do not have any object-code in the text segment; the text and data segments are merged in a manner similar to absolute programs. Unlike absolute programs, no-text programs do not have separate load records for each module.

No-text program is a special case of a shared text program. Since the size of the text segment is forced to zero for a no-text program, the next 4K boundary at which the data segment would be placed is location zero. The object-code generated by a no-text program is therefore identical to the same program linked to be an absolute program with an IPLA of 0.

The following memory map illustrates how the individual segments are placed in relation to other segments and modules in a no-text program. The module numbers are the order in which they appear on the command line; "m" is the last module specifed. Common blocks 1-x and library modules 1-n which are loaded to complete the program are also represented.

```
0 --> Text of mod 1
      Data of mod 1
      Text of mod 2
      Data of mod 2
            .
            .
      Text of mod m
      Data of mod m
      Text of library 1
      Data of library 1
      Text of library 2
      Data of library 2
            .
            .
      Text of library n
      Data of library n
                     <--+
      Bss of  mod 1     !   This memory is allocated to
      Bss of  mod 2     !   the task by UniFLEX at the time
            .           !   execution begins.
            .           !
      Bss of  mod m     !
      Bss of common 1   !
      Bss of common 2   !
            .           !
            .           !
      Bss of common x   !
      Bss of library 1  !
      Bss of library 2  !
            .           !
            .           !
      Bss of library n  !
                     <--+
```

5. LOAD AND MODULE MAPS

5.1. Load Map

The "m" option controls the printing of the module and load maps. When selected, the load map will provide information as to the type of output produced, the length of the resulting output object-code module, the number of input modules, and the transfer address.

5.2. Module Map

Use of the "m" option also selects printing of the module map. The module map describes the load addresses and object-code length for each of the input modules. The load addresses for each of the segments (text, data, and bss) take on different meanings depending on the type of output file produced.

5.2.1. The Module Map of a Relocatable Module

When producing a relocatable module, both the relocating assembler and the linking loader do not "bind" or tie addresses to absolute locations; they are made relative to the base of the segment to which they refer. The following example assembled by the relocating assembler will illustrate this point.

```
 1                                  ext     pdata
 2
 3    0000                          text
 4 + 0000 8E    0000      start     ldx     #msg1      point to 1st message
 5 +>0003 BD    0009                jsr     sub1       print it
 6 + 0006 8E    000B                ldx     #msg2      point to 2nd message
 7 + 0009 BF    0009      sub1      stx     msgaddr    save message address
 8 X 000C BD    0000                jsr     pdata      print a message
 9   000F 39                        rts                all done
10
11   0000                           data
12   0000 4D 65 73 73     msg1      fcc     'Message 1',0
13   000B 4D 65 73 73     msg2      fcc     'Message 2',0
14
15   0000                           bss
16   0000                           rmb     9
17   0009                 msgaddr   rmb     2          message addr save area
18
19 +                                end     start
```

All of the segments start at address 0 (lines 3, 11, and 15). This is called the base address. Because of this, it is possible for two labels in different segments to have the same address (offset from the segment base). "Sub1" and "msgaddr" is an example of this occurrence. All labels defined in a segment are relative to its base address. For example, "sub1" is 9 bytes from the beginning of the text segment, and "msg2" has an offset of 11 bytes from the base of the data segment. Throughout the linking process, it is guaranteed that the distance between "start" and "sub1" will remain constant. No assumptions, however, can be made about the distance between two labels that reside in different segments.

To produce a relocatable module from several input modules, the linking loader must combine all like segments. In other words, all text segments are concatenated starting with the text segment of the first input module, followed by the text of the second module, and so on. By doing so, however, the base address of all modules except the first will be changed. The linking loader automatically adjusts any addresses which refer to symbols in these modules which have been "relocated".

Five modules were assembled separately to produce the load and module maps used throughout the remainder of this chapter. For clarity, the lengths of each segment of these modules as reported by the assembler are listed in this table.

	sys_calls	pdata	main	stnd_input	strings
Length of Text Segment:	0000	001B	0015	0000	0000
Length of Data Segment:	0000	0000	0000	0025	0000
Length of Bss Segment:	0000	0000	0000	0000	0000

These five modules were linked to produce a relocatable output module.

```
Errors detected.
  write in sys_calls unresolved.
  ind in pdata unresolved.
  term in main unresolved.

Load map.
  relocatable output file produced
  program length: 005A
  number of input modules: 5
  transfer addr: 001B

TEXT  DATA  BSS   MODULE NAME
0000  0000  0000  sys_calls
0000  0005  0000  pdata
001B  0005  0000  main
0030  0005  0000  stnd_input
0030  0005  0000  strings
```

Notice that all modules except "sys_calls" have been relocated (their base addresses changed). Studying module "main", we can tell that its text segment has been relocated from $0000 to $001B. We can also tell by just looking at the module map that "main"'s data segment has length zero. This is true because the base address for the data segment of main ($0005) is the same as the base address for the data segment of "stnd_input". Since the segments of these two adjacent modules have the same base address, the length of the segment of the previous module must be zero. We cannot ascertain the lengths of the segments of the last module listed in the module map, however.

The program length is the sum of the lengths of the segments in all modules, i.e. if all the segments were loaded contiguously (including bss) starting at zero, the program length is the address of the last byte of the last segment loaded. Since the transfer address does not indicate in which segment it is contained, it is conceivable that the transfer address could reside in any of the three segments.

5.2.2. The Module Map of an Absolute Program

When creating an absolute, executable program, the user can specify the initial program load address (IPLA), and all address expressions are relocated relative to the IPLA. The same modules used in the relocatable example above were linked with an IPLA of $100 to produce the following load map:

```
Load map.
  executable, absolute output file produced
  program length: 015A
  number of input modules: 6
  transfer addr: 0120

TEXT  DATA  BSS   MODULE NAME
0100  0100  015A  sys_calls
0100  0105  015A  pdata
0100  0120  015A  main
0100  0135  015A  stnd_input
0100  0135  015A  strings
0100  015A  015A  /lib/Syslib
```

As you recall, processing of the relocatable modules combines the text and data segments when producing an absolute, executable program. The effect of this can be observed in the module map by noticing that the lengths of the text segments of all the input modules has been forced to zero, thus making the load address for all the text segments the IPLA.

Also note that the system library was searched to successfully resolve references not defined internally. The transfer address now correctly indicates that execution will begin with module "main".

5.2.3. The Module Map of a Shared-Text Program

When producing a shared text program, the linking loader collects all the text segments from the relocatable input modules and loads them at location 0. All of the data segments are then placed on the next even 4K boundary after the end of the text segments. Memory for the bss segments will be allocated immediately following the data segments at the time the program is executed.

The following load and module maps illustrates these facts.

```
Load map.
  executable, shared text output file produced
  program length: 102A
  number of input modules: 6
  transfer addr: 001B

TEXT  DATA  BSS   MODULE NAME
0000  1000  102A  sys_calls
0000  1005  102A  pdata
001B  1005  102A  main
0030  1005  102A  stnd_input
0030  1005  102A  strings
0030  102A  102A  /lib/Syslib
```

The base of the data segment ($1000) is the next even 4K boundary after the end of the text segment. Note that memory between the end of the text segment and $1000 is allocated to the task but never used. This would be an extreme waste of system resources.

5.2.4. The Module Map of a No-Text Program

Like an absolute program, the text and data segments are combined in a no-text program. A no-text program has the same result as an absolute program with an IPLA of 0.

The following load and module maps was produced by linking the same relocatable modules as the previous examples to produce a no-text program.

```
Load map.
  executable, no-text output file produced
  program length: 005A
  number of input modules: 6
  transfer addr: 0020

TEXT  DATA  BSS   MODULE NAME
0000  0000  005A  sys_calls
0000  0005  005A  pdata
0000  0020  005A  main
0000  0035  005A  stnd_input
0000  0035  005A  strings
0000  005A  005A  /lib/Syslib
```

Since the text segments are combined with the data segments, the text segment addresses are all zero.

6. MISCELLANEOUS

6.1. Transfer Address

A transfer address is the location at which execution is to start when the program is invoked. Use of the 'end' directive in the relocating assembler can be used to indicate a transfer address.

Only one relocatable module to be included in a program should contain a transfer address. If more than one module has a transfer address, the linking loader will accept the first one encountered and ignore all others.

6.2. Setid Bit and Executable Programs

The setid bit in the permissions field of a program temporarily gives any user of the program the same permissions which the owner of the program has. The 'setid' directive of the relocating assembler allows the programmer to turn on the setid bit in a relocatable module.

If any module is assembled using the 'setid' directive, the resulting executable program produced by the linking loader will have the setid bit turned on in the permissions field of the program.

7. ERROR MESSAGES

Attempt to redefine entry point <entry point name>
Two entry points (global symbols) exist with the same name. Entry point names must be distinct. One of the globals will have to be renamed, its module re-assembled, and linked.

Can't open '<file name>'.
An error was detected while trying to open the relocatable input module "file name". Possible causes are the file does not exist, the path leading to the file cannot be searched, or a read error was detected.

Can't open library '<file name>'.
Library "file name" cannot be opened. See above description for possible causes.

Conflicting options specified.
Options "a", "n", "r", and "t" are mutually exclusive, i.e. only one of them can be specified.

<entry point name> in <module name> unresolved.
An unresolved external has been detected by the loader. If the output of the loader is to be a relocatable module, this condition is probably expected and can be considered a warning. Use of the "u" option will suppress this warning message. However, when producing an executable program, this is an error.

<file name> - Contains assembly errors.
A relocatable input file to the loader contained errors when assembled. Correct the errors in the source code and re-assemble.

<file name> - Illegal input file.
The file name mentioned is not a relocatable module as built by the relocating assembler or the loader.

'Go' denied.
In order for the loader to execute the user's program, all of the following conditions must hold: an output file be produced and must be executable, the program must have a transfer address, and the program must be error free.

'Go' pending. Warnings detected. Still go (y or n)?
Warnings were detected by the loader while preparing a program for execution. The operator must answer with "Y" or "y" if these warnings are to be ignored and execution initiated; any other response will cause execution is to be aborted.

Illegal load address specified.
    The address following the "a" option is not a valid hexadecimal address. See the "a" option for complete specifications.

<file name> - Invalid library specified.
    The file name mentioned is not a valid library as built by the library generator program. All libraries must be created using the library generator.

Linking error at $<address> in <seg> segment of <module name>
    When adding the address of an external to a field being linked, the carry bit was set. "Module name" is the offending input module, and "address" is the offset of the link field from the base of the "seg" segment. This message is merely a warning to the user that an error condition may have occurred. In some cases, this message would be expected. Consider the following module:

```
                                    name    exttest
                                    ext     extrn
  0000                              text
X 0000 8E    FFFF                   ldx     #extrn-1
X 0003 C6    FF                     ldb     #extrn+255
                                    end
```

    The loader would report the following if the address of extrn was non-zero:

```
Errors detected.
  linking error at $0001 in text segment of exttest
  linking error at $0004 in text segment of exttest
```

    This first message is informing us that the sum of the address of extrn and -1 cannot be held in 2 bytes (the carry bit was set). However, the address loaded into the X register when this instruction is executed would be exactly as we would expect: one less than the address of extrn. A similar situation would occur when we force one-byte linking as in the second instruction of the above module: the sum of the least significant byte of the address of extrn (if non-zero) and 255 will not fit into one byte (the carry bit was set).

Loader Aborted!
    Always preceded by another message, this indicates the loader encountered a situation from which it cannot recover. Correct the fatal situation as reported by the previous message.

<module name> - Object code too large to load.
    The remaining memory available to the loader task after the loader program and stack has been allocated is insufficient to hold the entire module. To correct this situation, break the module source code into several smaller modules and assemble separately.

Symbol table overflow.
No more room exists in the symbol table to insert any more symbols. The symbol table maximum capacity is 347 symbols. Symbols which are not required to be global should be made internal in order to free up more room in the table.

Too many 'common' input modules.
The number of unique common blocks cannot exceed 30.

Too many input modules.
The number of input relocatable modules exceeds 100. This number includes the total of all user modules, library routines, and common blocks. Link a subset of these modules and produce a larger relocatable module. This single large module can then be linked with the remaining ones.

Too many libraries specified.
More than five libraries were specified on the command line. See the chapter LIBRARIES for further information.

UniFLEX error <error>
A UniFlex system error has been detected. Consult the "System Errors" section of the "Introduction to UniFLEX System Calls" manual.

Unknown option specified.
An invalid option was specified on the command line. Consult "INVOKING THE LOADER" chapter for valid options.

8. ETEXT, EDATA, AND END

In certain applications, it is desirable to know the last location contained in a particular program segment (text, data, or bss). Due to the manner in which the modules are loaded, it would be very difficult to determine these locations in an application program. To alleviate this difficulty, the loader has three global symbols which are always available and contain the location of the end of a segment. These three globals are ETEXT, EDATA, and END; corresponding to the ends of the text, data, and bss segments respectively.

ETEXT, EDATA, and END may be used like any other user defined global symbol. Since they behave like user defined globals, they will always appear in the Global Symbol Table listing. When used in a module, they should be defined as external. These special symbols are pre-defined, so users should refrain from using globals of his own with these same names.

This is the UniFLEX 6809 Relocating Assembler.

SYNTAX

```
relasmb file ...  [+options]
```

DESCRIPTION

The assembler accepts one or more input source files from the calling line and, according to options specified, produces an assembled source listing and/or a binary output file. There are basically two types of options. The first is single character options of which more than one may be supplied per plus sign:

+a	print absolute address on relative branches
+b	do not create binary output file
+d	do not print date in page header
+e	suppress printing of error summary information
+f	do not perform auto-fielding of source
+F	turn on debug or Fix mode
+g	list all code generated by fcb fdb, and fcc
+i	append all internal symbols to output binary module
+l	produce assembled listing output
+n	include decimal line numbers in listing
+s	produce printing of sorted symbol table
+S	limit the uniqueness of symbols to six characters
+u	make all undefined symbols externally defined
+w	suppress warning messages
+x	produce cross-reference information file
+0000	set number of symbols the symbol table accepts

Obviously the last option listed above is not a single character, but rather a decimal number. The second type of option is a single character, followed by an equals sign, followed by some parameter:

+a=prm	command line parameter a
+b=prm	command line parameter b
+c=prm	command line parameter c
+o=file	specify output binary file name
+p=000	specify starting page number for listing

Some valid examples follow:

```
relasmb /usr/john/sample
relasmb equates file* +bna
relasmb dumptxt +o=dump +ls +2400
```

SEE ALSO

UniFLEX 6809 Relocating Assembler User's Manual

This is the UniFLEX Linking Loader.

SYNTAX

```
     link-edit file ...  [+options]
```

DESCRIPTION

The linking loader accepts as input previously assembled, relocatable object-code modules and produces as output:

     1) A link edited, relocatable, object-code module, or
     2) a link edited, relocated, executable program.

The linking loader accepts two types of options. The first is single character options of which more than one may be supplied per plus sign:

```
     +i include all internal symbols in the symbol table
     +L do not search the libraries for unresolved externals
     +m print the load and module maps
     +n output is to be a no-text, executable program
     +p selects pagination of the printed output
     +r output is to be a relocatable, object-code module
     +s print the global symbol table
     +t output is to be a shared text, executable program
     +u do not print "unresolved external" message
```

The second type of option is a single character, followed by an equals sign, followed by a parameter:

```
     +a=addr      output is to be an absolute, executable program
     +g=command   execute the resultant program
     +l=library   specify a user library
     +N=mod name  specify a module name
     +o=file      specify an output file name
```

Some examples follow:

```
     link-edit main.r pdata.r
     link-edit mod[1-5].r +l=my_lib +a=100
     link-edit main.r sub1.r sub2.r +msN=phones
     link-edit payables.r +nmso=ap +l=/usr/john/library
```

SEE ALSO

UniFLEX Linking Loader Manual