Z-80 LINKING LOADER REFERENCE MANUAL

CONTROL DATA®
MP-32
COMPUTER SYSTEMS
Z80LDR Control Card Format

The Z80 Cross Loader (Z80LDR) is invoked by the following Control Card:

*Z80LDR(I=10,L=20,H=22)*

The table below describes the defaults and ranges of the various parameters. Parameters may be omitted, may stand alone, or may be equated to a numeric value in the range shown.

<table>
<thead>
<tr>
<th></th>
<th>ABSENT</th>
<th>ALONE</th>
<th>=XX</th>
<th>INPUT</th>
<th>LISTING</th>
<th>ABSOLUTE HEX OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>63</td>
<td>56</td>
<td>1-63</td>
<td>INPUT</td>
<td>LISTING</td>
<td>ABSOLUTE HEX OUTPUT</td>
</tr>
<tr>
<td>L</td>
<td>62</td>
<td>62</td>
<td>1-62</td>
<td>LISTER</td>
<td>LISTER</td>
<td>ABSOLUTE HEX OUTPUT</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>8</td>
<td>1-60</td>
<td>LISTER</td>
<td>LISTER</td>
<td>ABSOLUTE HEX OUTPUT</td>
</tr>
</tbody>
</table>

All values above are logical unit numbers. The Absolute Hex Output is in INTEL format.

ABNORMALITIES:

1. Z80LDR produces 2 extra lines of output before starting the Absolute Hex output. The lines consist of 2 dollar signs followed by 2 blanks. Most processors of INTEL hex format absolute loads will ignore lines that do not start with colon (:) and so the extra output is not serious.
FLEET NUMERICAL WEATHER CENTRAL
CONSOLIDATED COMMUNICATIONS SYSTEM
# Revision Record

<table>
<thead>
<tr>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Manual released.</td>
</tr>
<tr>
<td>(10-01-79)</td>
<td></td>
</tr>
</tbody>
</table>

Publication No.
CCS-A00X-02

Revision letters I, O, Q and X are not used.

Address comments concerning this manual to:

**Control Data Corporation**
Professional Services Division
205 Montecito Avenue
Monterey, Ca. 93940

© 1979
Control Data Corporation
Printed in the United States of America
LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

<table>
<thead>
<tr>
<th>Page</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cover</td>
<td>—</td>
</tr>
<tr>
<td>Title Page</td>
<td>—</td>
</tr>
<tr>
<td>ii thru v</td>
<td>A</td>
</tr>
<tr>
<td>1-1</td>
<td>A</td>
</tr>
<tr>
<td>2-1 thru 2-4</td>
<td>A</td>
</tr>
<tr>
<td>3-1 thru 3-19</td>
<td>A</td>
</tr>
<tr>
<td>4-1 thru 4-18</td>
<td>A</td>
</tr>
<tr>
<td>A-1 thru A-3</td>
<td>A</td>
</tr>
<tr>
<td>B-1 thru B-2</td>
<td>A</td>
</tr>
</tbody>
</table>

CCS - A00X - 02 Rev A
INTRODUCTION

This manual describes Microtec's Z80 Linking Loader that accompanies the Z80 Relocatable Assembler. The Linking Loader can be used to combine several independently assembled relocatable object modules into a single absolute object module. External references between modules are resolved with the final absolute symbol value being substituted for each reference.

The Loader not only provides for the linking of several modules and adjusting of the relocatable addresses into absolute addresses, but allows the program segment addresses to be specified, PUBLIC symbols to be defined, final load address to be specified and the order of loading of the program segments.
LOADER OPERATION

Many programs are too long to assemble as a single module. These programs can be subdivided into smaller modules and assembled separately to avoid long assembly time or to reduce the required symbol table size. After the separate program modules are linked and loaded by this program, the output module functions as if it had been generated by a single assembly.

The primary functions of the Linking Loader are as follows:
1. Resolve external references between modules and check for undefined references (linking)
2. Adjust all relocatable addresses to the proper absolute addresses (loading)
3. Output final absolute object module

To understand the loading process and to enable the user to use the Assembler and Linking Loader (hereafter called Loader) effectively, the user should understand the various program segments and segment load addresses. Although described in the Assembler Manual, the various segments are summarized below.

Absolute Segment - this is that part of the assembly program that contains no relocatable information but is to be loaded at fixed locations in the user's memory. Absolute code is placed into the object module exactly as it is read in the input modules.

Code Segment - the code segment contains that part of the program which comprises actual machine instructions and which typically can be placed into ROM. Instructions in the code segment can make reference to any other segment.
Data Segment - the data segment contains specifications for that part of a user's program that typically contains run time data and which usually resides in RAM. Of course, this segment could contain actual machine instructions.

Stack Segment - the stack segment is used as the Z80 run time stack during program execution.

Memory Segment - the memory segment is usually the high address portion of memory which is not allocated to any of the other segments. Data tables may expand into the memory segment but the assembler has no facility to cause instructions to be loaded into the Memory Segment. The start of the Memory segment is determined at Load time.

The Loader allows the user to load the program segments into a contiguous program module or to specify the starting address of any or all of the segments. The user may also specify the order in memory in which the segments will be placed. The default memory organization used by the Loader is shown below.

<table>
<thead>
<tr>
<th>High addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY BASE</td>
</tr>
<tr>
<td>STACK BASE</td>
</tr>
<tr>
<td>DATA BASE</td>
</tr>
<tr>
<td>CODE BASE</td>
</tr>
</tbody>
</table>
This is the typical memory organization used in most programs. Many users will want to place the STACK segment after the CODE segment so that the DATA segment can expand into the MEMORY segment during program execution.

The BASE address for all segments except the STACK segment is the low address of the segment. When a user specifies the starting address of a segment via a Loader command, it is the BASE address that is being specified. The BASE address for the STACK segment is the high address of the segment. This is done because during program execution the stack pointer typically moves toward lower addresses.

**Relocation Types**

The relocation type of any program segment is determined in the assembler by the CSEG and DSEG directives. The effect of the three relocation types in the Loader are explained below.

**Byte Relocation** - this implies that no operand was specified on the CSEG or DSEG directive. In this case the segment from the object module will be placed immediately after the same segment from the preceding object module and there will be no wasted memory.

**Page Relocation** - this relocation type is specified by the PAGE operand on the CSEG or DSEG directive in the Assembler. It implies that the program segment must begin on a page boundary (i.e. 0,100H,200H, ...). This code is placed by the Loader at the next available page boundary after the same segment type from the preceding object module.
Inpage Relocation - this is specified by the INPAGE operand on the CSEG or DSEG directive. It implies that the program segment must not cross a page boundary. If the loader determines that a program segment cannot fit within the current page, it begins the segment on the next page boundary as though it was PAGE relocatable.

In the typical load sequence, the Loader places all CODE segments contiguously in memory followed immediately by all DATA segments with no extra bytes between segments. However, if any of the DATA segments specify PAGE or INPAGE relocation then the Loader must start the DATA segment at a page boundary so that relocation will be preserved. To avoid any wasted memory the user can always specify starting addresses. In the above case the same problem exists if the DATA segment is followed by the CODE segment and the CODE segment has specified any PAGE or INPAGE relocation.

When initially developing and debugging a program it is helpful to specify each segment in each assembly as PAGE relocatable. This will then force that starting address of each module to end in 00H and will make it easier for the user to follow the flow of the program. In this case the assembler output listing contains the correct memory addresses except for an offset that must be added to the high order address byte. This may also be accomplished in the Loader by the CPAGE and DPAGE commands.
LOADER COMMANDS

The Loader reads a sequence of commands from the Command input device. The commands may be read in an interactive or batch mode (see Loader Installation Notes). The last command must be an EXIT or and END command.

The object modules are read from the object module input device or files specified on the LOAD command. The object modules may be read from the same input device as the commands.

The output of the Loader consists of an absolute load module suitable for loading into an actual microcomputer. The output module is written to the object module output device and is described in the Loader Installation Notes.

All commands begin in column 1. Command arguments may begin in any column and must be separated from the command by at least one blank. Comments may be placed in the command stream and are indicated by an asterisk in column 1.

The following pages describe the Loader commands. In the command descriptions, brackets { }, are used to indicate optional arguments. A summary of the commands is given below.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Set Code Segment Base Address</td>
</tr>
<tr>
<td>DATA</td>
<td>Set Data Segment Base Address</td>
</tr>
<tr>
<td>STACK</td>
<td>Set Stack Segment Base Address</td>
</tr>
<tr>
<td>MEMORY</td>
<td>Set Memory Segment Base Address</td>
</tr>
<tr>
<td>CPAGE</td>
<td>Set Paging for Code Segment</td>
</tr>
<tr>
<td>DPAGE</td>
<td>Set Paging for Data Segment</td>
</tr>
<tr>
<td>ORDER</td>
<td>Specify Segment Order</td>
</tr>
<tr>
<td>START</td>
<td>Specify Starting Output Module Address</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>STKLN</td>
<td>Specify Stack Length</td>
</tr>
<tr>
<td>NAME</td>
<td>Specify Output Module Name</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load specified Object Modules</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Specify PUBLIC symbols</td>
</tr>
<tr>
<td>LIST</td>
<td>List specified elements</td>
</tr>
<tr>
<td>NLIST</td>
<td>Do not list specified elements</td>
</tr>
<tr>
<td>EXIT</td>
<td>Exit Loader</td>
</tr>
<tr>
<td>END</td>
<td>End command stream and finish final load</td>
</tr>
</tbody>
</table>

* Comment

Command arguments that are numeric may be either decimal or hexadecimal. Hexadecimal constants are terminated by a \( H \), e.g. \( 1FH \), and need not have a leading zero if it starts with \( A-F \).

Commands may be read in any order and the same command may be used more than once. The last use of a command determines the command parameters. Commands may be placed before or after the LOAD command except for the CODE, DATA, STACK, and MEMORY commands, which if specified must precede the first LOAD command.
CODE — Set Code Segment Base Address

The CODE command is used to specify the starting address of the Code Relocatable Segments. If not specified, the starting address is zero or begins after the preceding segment if this is not the first segment in memory.

Example:

```
CODE 400H
```

where:

value — specifies the starting address of the CODE segment
DATA — Set Data Segment Base Address

The DATA command is used to specify the starting address of the Data Relocation Segments. If not specified, the starting address follows the CODE segment or is zero if the DATA segment is the first segment in memory.

Example:

```
DATA 1000H
```

where:

```
DATA value
```

value — specifies the starting address of the DATA segment.
STACK — Set Stack Segment starting Address

This command is used to specify the starting address of the STACK segment. The length of the STACK segment is specified by the STKLN command or is contained in the Load Module. If the Stack address is not specified, it will start immediately following the preceding segment in memory or begin at zero if this is the first segment.

Note that the BASE address specified by this command is the high address of the Stack Segment.

Example:

STACK 3FFH

where:

value — specifies the starting address of the STACK segment.
MEMORY — Set Memory Segment Base Address

The MEMORY command is used to specify the starting address of the MEMORY segment. The length of the MEMORY segment will be specified as zero on the load map but it is actually the length of available memory remaining in a user's system after the other segments have been loaded. If not specified, the starting address will start immediately following the preceding segment in memory or begin at zero if this is the first segment.

Example:

MEMORY 8000H

where:

value — specifies the starting address of the MEMORY segment.
CPAGE — Set Paging for Code Segment

This command may be used to modify the relocation type of code segments in the input object modules. As explained under Relocation Types (page 2-3), the assembler indicates to the Loader the relocation type as byte, page, or inpage for each segment in each object module. This command allows the user to override that relocation type specified by the assembler.

The typical use of this command is to allow the user to begin each module on a page boundary for ease of debugging and then to specify the final program as byte relocation to avoid any wasted memory space. This command allows the user to avoid reassembling each module and changing only the relocation type.

This command allows the user to specify the code segment of each module to be byte or page relocatable regardless of the type of relocation specified by the assembler. Inpage relocation is not affected. Note that the command also allows the relocation type specified by the assembler to be used by the loader. This command may be changed for each module read by the Loader. The last CPAGE command will be used.

Example:

```
CPAGE ON
```

CPAGE {blank, ON, OFF}

where:

blank — specifies that the relocation type will be that specified in the Assembler. This is the Loader default.

ON — specifies that the code segment of successive modules will be placed on a page boundary.

OFF — specifies that the code segment of successive modules will be adjusted to byte relocation.
DPAGE — Set Paging for Data Segment

This command may be used to modify the relocation type of data segments in the input object modules. This command is used in the same way as the CPAGE command and allows the user to specify the data segment of each module to be byte or page relocatable regardless of the type of relocation specified by the assembler. Inpage relocation is not affected.

This command may be changed for each modules read by the Loader. The last DPAGE command will be used.

Example:

```
DPAGE OFF
```

```
DPAGE {blank,ON,OFF}
```

where:

- blank - specifies that the relocation type will be that specified in the Assembler. This is the Loader default.
- ON - specifies that the data segment of successive modules will be placed on a page boundary.
- OFF - specifies that the data segment of successive modules will be adjusted to byte relocation.
ORDER — Specify Segment Order

As described under Loader Operation, the normal order of the segments in memory is: CODE, DATA, STACK, MEMORY. The ORDER command is provided for users who do not need to specify starting addresses for each segment but would like the segments to be placed in memory in a different order. If the user specifies starting addresses for the segments, the order of the segments if of no particular importance. If the user specifies starting addresses for only some of the segments, the remaining segments will be placed in the order specified by this command.

Example:

```
ORDER C,S,D,M
```

would place segments in the order CODE, STACK DATA and MEMORY

```
ORDER seg,seg,seg,seg
```

where:

seg — specifies one of the four segment types as follows:

- C — CODE
- D — DATA
- S — STACK
- M — MEMORY

all four segment types must be included in the command.
START — Specify Starting Output Module Address

This command is used to specify the starting address to be placed in the terminator record of the object module. If not specified the starting address is obtained from the END record of the main program of the input object modules. If no main program has been read, the starting address will be zero.

Example:

```
START 8
```

```
START value
```

where:

value - specifies the starting address to be used in the object module.
STKLN — Specify Stack Length

The STKLN command is used to specify the length of the STACK segment to the Loader. If not specified, the stack length is determined by the sum of the stack segment lengths specified in the load modules.

Example:

    STKLN 20H

STKLN value

where:

value — specifies the length of the STACK segment.
NAME — Specify Output Module Name

The NAME command is used to specify the name of the final output object module. Currently this command performs no function for the output module as the module is in Intel's hexadecimal format and contains no name. It will be used when the output object module is in relocatable format. The user specified name may be any standard symbol and up to 6 characters. If the user does not specify a name, the name of the output module will be taken from the first input module.

Example:

NAME READER

NAME name

where:

name — is a symbol that specifies the object module name
LOAD — Load specified Object Modules

The LOAD command is used to specify one or more input object modules to be loaded. If the command operand is a number, it is assumed that the input module is to be read from that logical I/O device. If the command operand is not a number, it is assumed that the name of a disk file is being specified and the object module will be read from the file. If any operand is preceded by a minus sign, it indicates that the object modules should be read from the specified device or file until an end-of-file condition (EOF) is detected (see Installation Notes concerning modifications for EOF). In this case the user need not specify an operand for each object module.

Object modules may be read from a combination of files and peripheral devices and may or may not be read until the EOF. The object modules are loaded in the order specified with each module being loaded into memory at a higher address than the preceding module. A user may use as many LOAD commands as needed.

Example:

```
LOAD 7,-FILE1,7
```

Four modules are to be loaded. The first from unit 7, the next two from FILE1 until an EOF, and finally the last from unit 7.

```
LOAD module1,module2,...,module4
```

where:

- `module1` — specifies the number of a logical input device or the name of a disk file on which the object module resides. Any module specification preceded by a minus sign will read object modules until an EOF is detected on the device or file. Operands are separated by commas.
PUBLIC — Specify PUBLIC Symbols

This command is used to define and/or change the value of a PUBLIC symbol. If the symbol specified by this command is already a PUBLIC symbol (from an object module), the value of the symbol is changed to that specified by the user. If the symbol specified by this command is not already defined, it will be entered in the Loader Public symbol table along with the specified value and will then be available to satisfy external references from object modules.

This command is useful in that it allows the user to specify the value of some external symbols at load time and possibly avoid any reassembly. To change the value of a symbol that is PUBLIC in an object module, this command must be specified after the object module has been loaded via the LOAD command.

Example:

PUBLIC INPUT=2FH,OUTPUT=200H

PUBLIC sym₁=val₁{,sym₂=val₂, ...,symₐ=valₐ}

where:

symᵢ — is user defined PUBLIC symbol which may already be defined by some object module.

valᵢ — is the value given to the symbol.
LIST — List Specified Elements

The LIST command may be used to generate listings of the elements specified. The defaults are: no symbol tables are listed, an object module is produced, no symbols are placed in the output object module, and local symbols are not purged from the input modules. The user should note that placing both PUBLIC and local symbols into the output object module symbol table could cause duplicate symbols to exist in the module. Typically only the local symbols placed into the object modules by the assembler or PUBLIC symbols will be placed into the output object module. This is the case since using the B option in the assembler forms a symbol table which includes PUBLIC symbols.

Example:

LIST T,X list both local and PUBLIC symbol tables

LIST D,O,P,S,T,X

where:

D — specifies that PUBLIC symbols will be placed into the output object module.
O — specifies that an object module is to be produced. (default)
P — specifies that any symbols present in the input modules be placed into the Loader symbol table. (default)
S — specifies that the local symbol table be written to the object module and thus may be used for debugging.
T — specifies that the local symbol table be listed on the list output device.
X — specifies that the PUBLIC symbol table be listed on the list output device.
NLIST — Suppress Listing of the Elements Specified

The NLIST command is the opposite of the LIST command and is used to suppress the listing of the elements specified. The elements may be turned back on with the LIST command.

Example:

```
NLIST 0  don't produce an object module
```

```
NLIST  D,O,P,S,T,X
```

where:

D — specifies that PUBLIC symbols will not be placed into the output object module. (default)

O — specifies that no output module is to be produced. This is useful to check for errors.

P — specifies that any local symbol tables present in the input modules not be placed in the Loader symbol table. This is useful if many modules are being loaded and the symbol table may become full. Of course these local symbols may then not be listed in a symbol table.

S — specifies that the local symbol table not be written to the object module. (default)

T — specifies that the local symbol table not be listed on the list output device. (default)

X — specifies that the PUBLIC symbol table not be listed on the list output device. (default)
EXIT — Exit Loader

The EXIT command is used in the interactive mode to exit the Loader. This command is useful when the user finds an error that will require the exiting of the Loader to fix. It acts like an END command except the final load does not take place and an output object module is not produced. This command may also be used in the batch mode by making it the last command in the command stream. In this case the final load will not take place but the object modules and commands will be read and checked for errors.
**END** — End command stream and finish final load

The **END** command should be the last command in every Command stream except if the **EXIT** command is used. It initiates the final steps in linking and loading the input modules. An exit is then made from the program.
Comment — Specify Loader Comment

An asterisk may be used to specify a comment in the command input stream. The asterisk should be in column one.

Example:

* SAMPLE LOADER PROGRAM
HOW TO USE THE LOADER

The Loader

The loader program is usually supplied as an unlabeled unblocked magnetic tape with 80 character card image records. Other media may be requested.

The Loader is written entirely in Fortran and is comprised of a main program and several subroutines. The main program appears first on the tape the last subroutine is followed by a tape mark. The Loader is located after the assembler and assembler test program on the tape.

The Loader Installation Notes describe program installation and any modifications that may have to take place for a particular computer. It is extremely helpful to read these notes before installing the program.

Loader Execution

This is a two pass Loader in which the commands and object modules are checked for errors during the first pass and a symbol table of PUBLIC symbols is formed. Errors detected during this phase of the program will be displayed on the listing. If the user is in batch mode, any errors found during this pass will cause the loader to terminate with the message "LOAD NOT COMPLETED". If the user is in interactive mode, only those errors found in the object modules will cause termination of the loader.

During pass two of the Loader the final object module is produced and any undefined externals are printed on the list device along with their address in the object module. A symbol table may also be listed.
When executing the Loader, the user should place the Loader Commands on the command input device expected by the program. Of particular importance is that the user specify the correct number of modules to be loaded and where they are loaded from on the LOAD command. It is extremely useful to use the read until EOF option on the LOAD command if the end-of-file can be detected on the particular computer.

Loader Listing

The following pages show a sample listing from the Loader which is used to describe both the output listing and the Loading process. This example is also used as the Loader Test Program.

The first page of the output listing lists all commands entered by the user along with any command errors that occur. Following this would be any load module errors that occurred in the modules loaded via the LOAD command. If no fatal errors occur up to this point, then a load map is displayed which lists the names of all input modules followed by the starting addresses of the CODE and DATA segments for that module. The ending address+1 for each segment is displayed at the end of all modules and is indicated by //.. Following this, the starting and ending addresses of the STACK and MEMORY segments are displayed. The ending addresses plus one are once again shown by the double slashes. When the starting and final addresses are the same, it implies that the length of the segment is zero. Following this is a list of all absolute segments in the object module along with the starting and ending addresses. It is possible that all absolute segments will not be shown if certain Loader tables become full.

Following the Load Map is a list of all PUBLIC symbols as well as local symbols if the user specified the appropriate
LIST command. PUBLIC symbols are those declared public in the assembler by the PUBLIC directive. Local symbols are those that were output by the assembler if the user had specified the "LIST B" directive in the assembler. These may be used for debugging but serve no function to the Loader.

As shown on the example listing, the only other information that will be displayed on the listing after this point are any undefined externals found during final load. This is indicated by the name of the module that contains the undefined external, the address of the undefined external in the input object module, the segment type and name of the external.

The end of the Load program is indicated by the "LOAD COMPLETED" or "LOAD NOT COMPLETED" message.

**LOAD LR COMMANDS**

**TEST PROGRAM FOR Z80 LOADER**

**NOTE THE OBJECT MODULES ARE READ IN FROM THE SAME**

**DEVICE AS THE COMMAND STREAM. TO READ THE OBJECT MODULE**

**FROM A DIFFERENT DEVICE THE LOAD COMMANDS MUST BE**

**CHANGED TO THE NEW DEVICE NUMBER. ALSO IF THE USERS**

**COMMAND DEVICE IS NOT 5, THE LOAD COMMANDS MUST ALSO BE**

**CHANGED.**

**LIST 1.S.X**

**DATA 447H**

**CODE 605H**

**ORDER CS,DS,DS**

**STACK 400H**

**STKLN 12**

**LOAD 5,5**

**LOAD 5**

**END**

**LOAD MAP**

<table>
<thead>
<tr>
<th>MODULE</th>
<th>CODE</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>0605</td>
<td>6407</td>
</tr>
<tr>
<td>READ</td>
<td>063F</td>
<td>6458</td>
</tr>
<tr>
<td>MODULE</td>
<td>0693</td>
<td>6580</td>
</tr>
<tr>
<td></td>
<td>06A4</td>
<td>06F0</td>
</tr>
<tr>
<td>STACK</td>
<td>09F4</td>
<td></td>
</tr>
<tr>
<td>MEMORY</td>
<td>050F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>050F</td>
<td></td>
</tr>
</tbody>
</table>

**ABSOLUTE SEGMENTS**

|        | 0000 | 000F |

CCS-A00X-02 Rev. A  4-3
**PUBLIC SYMBOLS**

CRLF  L634  ECHO  J457  IEUFEN  L457  INBUF  L467
READ  L63F  TIN  L61C  TOUT  L624

**LOCAL SYMBOLS**

ASCR  L60D  BLNK  L02W  BSFA  L008  READ  L63F
READ1C  L644  READ2U  L552  READ3U  L65F  READ4U  L669
READ5U  L673  READ6U  L670  READ7U  L68D  READ8U  L686
TAB  L008

**MODULE MAIN**

UNDEFINED EXTERNALS

6611 C - SCAN

**LOAD COMPLETED**

LOADER EXAMPLE

```
$$
ASCR  ULCUH  ULKU  U62H  BSFA  U68H  READ  U63F
READ1C  U66H  READU  U56H  READ3U  U65F  READ4U  U669
READ5U  U67H  READ6U  U67H  READ7U  U68D  READ8U  U686
TAB  U008
$$
```
Loader Example

The following pages show three assembly listings of programs that will be combined by the Loader. The actual Load is shown on the preceding pages. The main program contains references to a subroutine READ and SCAN which are not in the program but are declared external and will be found in another object module. The second assembly listing shows the READ routine which is required by the Main program and also shows that the READ routine requires I/O drivers TIN and TOUT which are declared external and will be found in the Main program. The third program contains no links to the other programs but does contain some absolute code which will be used for a RST instruction during execution.

The Command stream on the preceding pages shows that the user has specified the starting addresses of both the CODE and DATA segments in addition to changing the order of the segments to CODE, STACK, DATA, and MEMORY. The LIST command is then used to obtain a symbol table listing of both local and PUBLIC symbols as well as placing the local symbols into the output object module. Finally the LOAD command is used to read the three modules from the device shown.

The Load Map shows the starting and ending addresses of the three modules in the order loaded. Note that the third module had specified a "DSEG PAGE" directive in the assembly and the load map shows that the data segment for this module indeed starts on the next page boundary.

An undefined external is listed for the Main module and its address, relocation type and name is specified. It can be seen that SCAN is not in any module. The user could have specified the address of the routine with a PUBLIC command.
Finally the symbol table of all PUBLIC and local symbols used in the program along with their absolute addresses is listed. The user can determine from the addresses as well as the final object module displayed on a subsequent page that the modules have indeed been linked together to form a final absolute module with all addresses adjusted to the correct value and any links between modules resolved.

Following the above example, a Loader run is displayed that contains a few errors. Most of the load errors shown will not occur except under unusual conditions and they have been shown for informative purposes only.

The final absolute object module from the example is shown along with the local symbols that were placed into the module.
LIST
NAME MAIN
PUBLIC INBUF,INUFEND,TIN,TJUT,JRLF,ECHO
EXTERN READ,SCAN

// THIS IS A SAMPLE PROGRAM THAT SHOWS MOST OF THE RELOCATABLE
// FEATURES OF THE ASSEMBLER. THREE MODULES ARE LINKED TOGETHER
// TO FORM THE FINAL PROGRAM. PUBLICS AND EXTERNALS ARE USED
// TO PERFORM THE LINK.

// BELOW IS THE MAIN PROGRAM AND I/O DRIVERS. THIS IS
// LINKED TO A ROUTING WHICH READS A LINE OF CODE AND WHICH
// ITSELF REQUIRES THE I/O DRIVERS.

CSEG

MAINT LD SP,STACK
CALL READ
LD ML,INBUF
LD A,(HL)
CP DL
INC HL
JP NZ,MAINT
CALL SCAN
INC HL
JP MAIN

NAME = IN8

// THIS ROUTINE WILL INPUT A CHARACTER FROM THE TERMINAL
// ENTRY PARAMETERS
// NONE
// EXIT PARAMETERS
// A = INPUT CHARACTER
// B = SAME AS A
// REGISTERS USED
// A,B

IN8 IN A,(USTAT)
AND KKDY
JP Z,IN8
IN A,(UDATIN)
AND L
LD H,A

NAME = OUT4

// THIS ROUTINE IS USED TO OUTPUT A CHARACTER TO
ENTRY PARAMETERS

- CHARACTER TO OUTPUT

EXIT PARAMETERS

NONE

REGISTERS USED

A, U

OUT81

IN

A,(USTAT)

TROY

AND

Z,OUT8

JP

LO

OUT

A,B

(UDATOUT), A

RET

NAME = CKLF

This routine outputs a carriage return and line feed.

CRLF1

LD

B,ASCR

CALL

OUT8

LD

B,ASLF

CALL

OUT8

RET

ISLF DATA SEGMENT

INPUT BUFFER

END OF BUFFER

ECHO FLAG

USART STATUS

USART OUTPUT

USART INPUT

TRANSMIT READY

READER READY

ASSEMBLER ERRORS = 0
<table>
<thead>
<tr>
<th>LABEL</th>
<th>VALUE</th>
<th>REFERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCR</td>
<td>440</td>
<td>74 - 49</td>
</tr>
<tr>
<td>ASLF</td>
<td>442</td>
<td>81 - 55</td>
</tr>
<tr>
<td>BLNK</td>
<td>442</td>
<td>22 - 96</td>
</tr>
<tr>
<td>CRLF</td>
<td>B2F</td>
<td>3 - 19</td>
</tr>
<tr>
<td>ECHO</td>
<td>435</td>
<td>3 - 88</td>
</tr>
<tr>
<td>IBUFEN</td>
<td>465</td>
<td>3 - 67</td>
</tr>
<tr>
<td>INB</td>
<td>447</td>
<td>44 - 46</td>
</tr>
<tr>
<td>INBUF</td>
<td>460</td>
<td>3 - 24</td>
</tr>
<tr>
<td>MAIN</td>
<td>466</td>
<td>-10 - 27</td>
</tr>
<tr>
<td>MAING</td>
<td>468</td>
<td>-21 - 24</td>
</tr>
<tr>
<td>MEMORY</td>
<td>400</td>
<td>9</td>
</tr>
<tr>
<td>OUTB</td>
<td>6024</td>
<td>-67 - 69</td>
</tr>
<tr>
<td>READ</td>
<td>404</td>
<td>4 - 19</td>
</tr>
<tr>
<td>RDDT</td>
<td>406</td>
<td>45 - 43</td>
</tr>
<tr>
<td>SCAN</td>
<td>404</td>
<td>4 - 25</td>
</tr>
<tr>
<td>STACK</td>
<td>440</td>
<td>4 - 25</td>
</tr>
<tr>
<td>TIN</td>
<td>441</td>
<td>3 - 97</td>
</tr>
<tr>
<td>TOUT</td>
<td>404</td>
<td>3 - 90</td>
</tr>
<tr>
<td>TRDY</td>
<td>410</td>
<td>66 - 92</td>
</tr>
<tr>
<td>UDATIN</td>
<td>400</td>
<td>47 - 91</td>
</tr>
<tr>
<td>UDATO</td>
<td>400</td>
<td>71 - 98</td>
</tr>
<tr>
<td>USTAT</td>
<td>400</td>
<td>44 - 87</td>
</tr>
</tbody>
</table>
26L ASSEMBLER VLS

NAME = READ

THIS ROUTINE READS A LINE FROM THE TERMINAL AND PLACES IT INTO THE INPUT BUFFER. THE FOLLOWING ARE SPECIAL CHARACTERS:

CR = END OF CURRENT LINE
CONTROL X = DELETE CURRENT LINE
DEL = DELETE LAST CHARACTER
ALL DISPLAYABLE CHARACTERS BETWEEN BLANK AND Z AND THE ABOVE SPECIAL CHARACTERS ARE RECOGNIZED BY THIS ROUTINE AS WELL AS THE TAB. ALL OTHER CHARACTERS ARE IGNORED. AN ATTEMPT TO INPUT MORE CHARACTERS THEN IS ALLOWED IN THE INPUT BUFFER WILL BE INDICATED BY A BACKSPACE.

ENTRY PARAMETERS
ECHO = ECHO FLAG, 0 = NO ECHO

EXIT PARAMETERS
INBUF = OBTAINS INPUT LINE

REGISTERS USED
A, B, E, H, L

INPUT BUFFER ADDRESS
SET CHARACTER COUNT
READ NEXT CHARACTER
CHECK FOR CONTROL X
NOT CONTROL X
START AGAIN
CHECK IF CR
INO
GET COUNT
CHECK IF ANY INPUT
KEEP READING
INPUT CR AT END OF LINE
CHECK FOR DELETE
NOT DELETE
GET COUNT
INO ENTRIES YET
DECREMENT COUNT
GET A BACKSPACE
OUTPUT BACKSPACE
ERR LINE  ADD  ul de HJ Bt
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

LIST
ASEG
OKG
PUSH ML
PUSH BC
LD ML1 (DATA)
JP 1.COM
CSEG
NOP
LD ML+8
NU  A, (DATA)
OK A
JP NZ+LAB1
CPL
LAB1: LD ML, DATA+3
HALT

PAGE
DSEG

JPG LAB1
LD BC, DATA
ADD A+B

DLFW LAB1
DEFW LOW+DATA
DATA: DLFB 56, +LOW+LAB1

NOP
END

ASSEMBLER ERRORS = 0
<table>
<thead>
<tr>
<th>LABEL</th>
<th>VALUE</th>
<th>REFERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>D 0000</td>
<td>6 11 16 21 25 26</td>
</tr>
<tr>
<td>LAPI</td>
<td>C 0000</td>
<td>13 -16 26 24 26</td>
</tr>
<tr>
<td>MEMORY</td>
<td>M 0000</td>
<td>0</td>
</tr>
<tr>
<td>STACK</td>
<td>S 0000</td>
<td>0</td>
</tr>
</tbody>
</table>
**LOADER COMMANDS**

* TEST PROGRAM FOR 20K LOADER

* NOTE THE OBJECT MODULES ARE READ IN FROM THE SAME
  DEVICE AS THE COMMAND STRING TO READ THE OBJECT MODULE
  FROM A DIFFERENT DEVICE THE LOAD COMMANDS MUST BE
  CHANGED TO THE NEW DEVICE NUMBER. ALSO IF THE USER'S
  COMMAND DEVICE IS NOT 5, THE LOAD COMMANDS MUST ALSO BE
  CHANGED.

LIST 1,5,X
DATA 447H
CODE 665H
ORDER C,5,0,0,M
STACK A,0,H
STKLN 12
LOAD 5,5

**MODULE MAIN**
  RECORD OUT OF SEQUENCE
  RECORD 5 - 0000000000000000
  LOAD 5

**MODULE**
  HEADER RECORD ERROR
  RECORD 1 - 0000000000000000C52A0B6C326147
  END

**LOAD NOT COMPLETED**
APPENDIX A

LOADER MESSAGES

Messages from the Loader may be classified into Command Error Messages and Load Messages. Command errors are due to invalid commands or command parameters and always cause termination of the Loading process in batch mode. Command messages are listed beneath the actual command on the output listing. Load messages occur during the loading of the object modules initiated by the LOAD command. These messages may be fatal or informative. For most load messages, the message is listed followed by the record number in the input module and the actual record in error. The module name is also listed at the start of the messages for a particular module.

Most load errors should not occur and if they do, the user is advised to first reassemble the program and attempt to reload.

Command Messages

Invalid Command - a command specified by the user is not a legal Loader command.

Invalid Operand - an operand specified for a command contains invalid characters, does not exist, or is too large.

Command Not Allowed - this command is not allowed at this point in the program. Due to specifying a load address after a LOAD command has been specified.

Symbol table Full - user specified a PUBLIC command and no more room exists in the symbol table.

Module Greater than 64K - at final load time the lengths of all program segments is greater than the 64K memory size.
File Not Found - a file specified in the LOAD command does not exist or possible an invalid LOAD command operand.

Invalid Symbol - a PUBLIC command is specified that contains an invalid symbol.

Load Messages

Invalid Hex Character - a character in the record shown contains an invalid hexadecimal character. Some records contain symbols as well as hexadecimal numbers. This message does not apply to those symbols in the record.

Invalid Checksum - the record has a checksum error and probably contains some changed characters.

Header Record Error - a header record was not the first record in the object module or a header record was found after the first record.

Record too large - a record specifies a record length that is greater than 72 characters.

Invalid Record Type - a record specifies a record type that does not exist in the Loader.

Invalid ID or type - some internal parameters on this record are invalid.

Address out of range - a relocation record specifies relocation at an address outside the range of relocation specified on the header record.

External Index out of Range - an External Reference is made to an external symbol that does not exist.

External Table Full - Current object module specifies more external symbols then may be contained in external table. Increase size of table.
Record out of sequence - an object module record was read that is out of sequence in the module or the user may have inadvertently mixed the records if they exist on cards.

Symbol Table full - a PUBLIC object module record is being processed and the symbol table is full.

Undefined External - a reference is made to an external symbol that has not been defined in another module or by the user. The name, relocation type, and address of the symbol in the original module is listed.

Duplicate PUBLIC Name - a PUBLIC symbol is defined that has already been defined in another module. Loading will continue and the PUBLIC name will be listed.

Module Greater than 64K - during initial loading, the sum of all segment lengths exceeds the 64K memory size.

Segment Overlap - due to user specified addresses, one or more of the segments overlap. This is an informative message and loading continues. An absolute segment could also overlap a relocatable segment.

Unexpected end of Module - the user has used the EOF option on the LOAD command and an end-of-file condition has occurred before the current module end record. Possibly some of the information in the load module is out of order or not in the load module. User should reassemble the module and check that device or file contains proper load modules. Program termination occurs for this error.
APPENDIX B

OBJECT MODULE FORMATS

As part of the output processing, the Loader produces an absolute object module. This object module is a machine readable computer output in the form of punched cards, paper tape, etc. The output module contains specifications for loading the memory of the target microprocessor.

The object module produced by the Loader uses the standard Intel hexadecimal format. This was done for a number of reasons. The object module in this format contains its own load address. The user may easily create their own object records for patches. This is the format used by some 280 manufacturers such as MOSTEK. Finally, the object module does not contain any special characters such as those used by the Zilog development system.

The object module is normally punched out on the device specified. However, through use of the NLIST and LIST directive the output module may be deleted.

The object module is produced as a series of card images on the output punch device. Each object record contains the load address and data specifications for up to 16 bytes of data. Symbol table information may also be included. The format of an object module is shown below.

$$
symbol records
$$
data records

A sample symbol record is shown below:

APPLE 00000H LABEL1 0D0C3H MEM OFFFFH

As many symbols records as needed may be contained in the object module.
module. At most 4 symbols per line are used but each line need not contain 4 symbols even if it is not the last line. A module may contain no symbol records in which case the "$$" records will still be contained in the module.

The format for a data record is shown below.

```
  1 2 3 4 5 6 7 8 9 10 11 ... 40 41 42 43
  : byte   load   type   data   data checksum
         count  address
```

Column 1 contains the code for a colon. This marks the beginning of an object data record.

Column 2 and 3 contain the count of the number of data bytes on the record. If this field contains an "00" it signifies the end of the object module.

Columns 4 through 7 contain the load address expressed as hexadecimal digits. The first data byte is to be loaded into this address, subsequent data bytes into the next sequential addresses. Columns 4 and 5 contain the most significant byte of the address.

Columns 8 and 9 contain the record type. Presently two types are defined. "00" indicates a data record. "01" indicates a terminator record. In this case the byte count will also be zero and the load addresses will actually be the starting address.

Columns 10 to 41 (or less if less data) contain the hexadecimal specifications for up to 16 bytes of data.

The last two columns in the record contain a checksum. The checksum is the negative of the sum of all bytes on the record (except column 1) evaluated modulo 256.