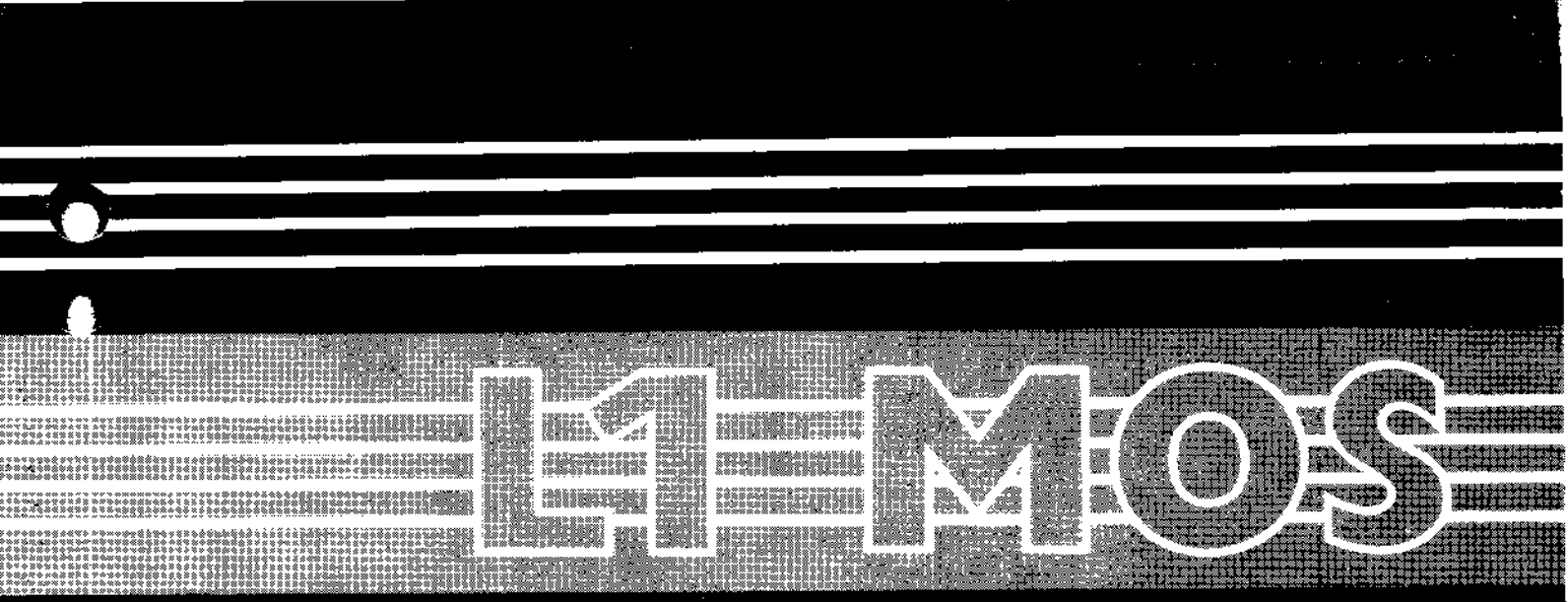

HMOS

L1MOS

Program Development Tools
Reference Manual

olivetti

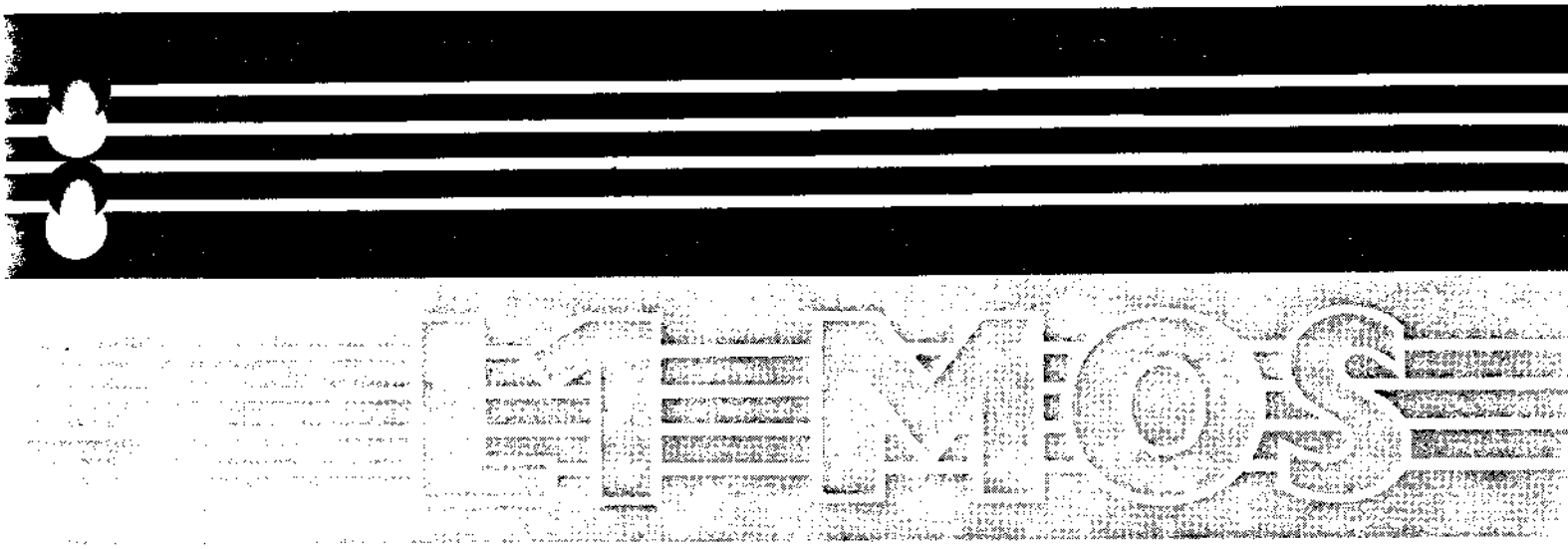
PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 IVREA (Italy)

UPDATING STATUS

LEVEL	DATE	UPDATED PAGES	PAGES	CODE
3	June 85		140	4002790 S (3)
4	Jan. 86	Pref., v÷x, PART I, 1-1, 1-9, 1-10 ÷ 1-12, 2-2, 2-6, 2-7, 2-12, 2-14, 3-4, 3-5, 3-9, 3-11, 3-11/B, 3-15, 3-16, 3-21÷3-22/A, 3-24, 3-28÷3-31, 3-36÷3-46, 4-3, 4-4, 4-6, 4-8, 5-3, 6-5, 6-6, 6-11, 6-19, 9-1÷9-4	57	4002794 Y

Pages marked * must be suppressed



Program Development Tools
Reference Manual

olivetti

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 IVREA (Italy)

PREFACE

This manual describes the generalised tools for program preparation and execution.

It is addressed to COBOL, BASIC, FORTRAN and Pascal+ programmers who wish to link and debug programs which run on the L1 MOS system.

It assumes a basic knowledge of COBOL, BASIC, FORTRAN and Pascal+ and of the MOS operating system.

SUMMARY

The manual is subdivided into three parts:

- Part I describes the OLINK Generalised Linker and the commands for its use.
- Part II describes the DEB Generalised Symbolic Debugger and the commands for its use.
- Part III describes other generalised utilities for program preparation.
  - The FFT utility for the transfer of files from the S6000 system to the L1 MOS system via a floppy disk.
  - The CHECK utility for a statistical report on object modules.
  - The M5LDUMP utility for the dump of a MOS l-module.

REFERENCES

To read first...

Introduction to MOS - Code 4002130 G (Vol. 2)

SHELL Commands - Reference Manual - Code 4002770 Q (Vol. 3)

COBOL - Program Preparation and Execution - Code 4004310 T (Vol. 6D)

Compiled BASIC - Program Preparation and Execution -
Code 4002180 M (Vol. 6E)

For further information, read...

PASCAL+ Program Preparation and Execution - Code 4002480 T (Vol. 9)

MOS - Programmer Guide - Code 4002570 L (Vol. 6G)

Glossary/Glossario - Code 4002140 H (Vol. 1)

FIRST EDITION: March 1984 - Release 3.0

UPDATES: August 1984 - Release 4.0
January 1985 - Release 4.1

REPRINTED: February 1985

(Including update 1)

SECOND EDITION: June 1985 - Release 5.0

UPDATES: January 1986 - Release 5.1
December 1986 - Release 5.2

CONTENTS

THE GENERALISED LINKER (OLINK)

THE GENERALISED SIMBOLIC DEBUGGER

GENERALISED UTILITIES

APPENDICES

PART I - THE GENERALISED LINKER (OLINK)

INTRODUCTION TO PART I

Part one of this manual illustrates how the compiled units which constitute a COBOL, BASIC, FORTRAN and PASCAL+ program are linked and allocated so that the program may be run.
We have first described the Linker characteristics and the general concepts which must be understood before a link operation may be effected. This is followed by a description of the "Linker call" and all Linker commands with examples.
The error message description follows.

1. CHARACTERISTICS

INTRODUCTION

The MOS Generalised Linker (OLINK) is aimed at producing load modules, which may be loaded into central memory, starting from the output of the COBOL, BASIC, FORTRAN and Pascal+ compilers. The format of these modules is such that they may be executed by the L1 system hardware, under the control of the MOS operating system.

The linker can produce both exclusive and shared modules. Shared modules are used by several processes to share procedures and constants.

The load-modules produced by the Linker may be either private or sharable. The latter allow procedures and constants to be shared by more than one process.

Each load module created by the Linker may have an associated entry-point table. This ensures that a load module may be used as input in a successive link operation.

The Linker assigns values and virtual addresses to the symbols defined within each module as well as to the external symbols which the module refers to. If a symbol has not been defined in a module which has been input to the Linker, the Linker may search for it in the sharable modules and/or in the libraries in order to find its references.

FUNCTIONS

Some or all of the following operations are executed by the Linker when it creates a load module. The order in which these operations are listed is only indicative and must not be considered a true reflection of the order in which they are actually executed. The Linker:

- Allocates a virtual memory with a logical address
- Builds overlaid program segments
- Resolves symbol references within object modules in input
- Checks for compatibility between symbol definitions and references
- Builds the load module file
- Generates listings and mappings
- Constructs libraries.

VIRTUAL MEMORY ALLOCATION

The Linker splits an object module into sections before storing it into the allocated virtual memory.

Section

Program sections are units which are produced by the code generators (compilers or generators) and used by the Linker to define and make reference to homogeneous and indivisible parts of a program (executable code, data, constants etc.). A program section is made up of a memory area having a name, length, alignment and a series of attributes which describe the permitted or the intended use of the memory area. The alignment specifies whether a memory area is to have its address aligned to a word or a double word, or in the case of absolute sections it indicates the number of the segment into which the section is to be loaded.

Segment

The memory segment is a hardware concept. The Linker uses segments to describe the memory requirements of the entire load-module to the Process and Memory Manager (PMM). A memory segment is made up of a contiguous memory area which is sharable and has the same protection characteristics as a hardware segment.

Each segment is identified by a number.
The Linker creates memory segments by putting together program sections which have similar (but not necessarily identical) attributes. The user may request the Linker to create segments via the appropriate command.

Section Attributes

A section may have the following attributes:

- Overlay / Concatenate
- Relocatable / Absolute
- Sharable / Private
- Executable
- Readable
- Writable
- Fetchable
- Interpreted
- Debug
- Linked
- Stack

A program section may have more than one of the above attributes concurrently.
The following is a brief description of each one of these attributes.

Overlay / Concatenate Section

This attribute tells the Linker how to allocate memory to sections, of different object modules, having the same name.
Linked sections require their own virtual memory allocation. The Linker allocates the sections having the same name in contiguous memory areas.
Overlaid sections share the virtual address space. The Linker allocates the sections having the same name in the same virtual memory area.

Relocatable / Absolute Section

This attribute tells the Linker whether the section should be assigned a virtual address space (relocatable) or not (absolute).

Sharable / Private Section

This attribute tells the Linker whether the program section is to be shared.

Executable Section

This attribute informs the Linker that the program section contains executable code statements.
The Linker will group all program sections that have this attribute but not the Readable or Writable attributes into a separate memory segment. This segment is known as an "execute-only" memory segment. The segment needs to be resident in memory only when a statement is executed and is then automatically loaded and unloaded by the operating system (not yet available).

Readable Section

This attribute informs the Linker that the contents of the program section are readable. Data sections are a typical example of Readable sections.
The Linker will normally attempt to build a memory segment of all program sections that have the Readable attribute but not the Executable or Writable attribute.

Writable Section

This attribute informs the Linker that the program section contains areas which the program may write to. The Readable attribute must also be set if the data area is also to be read.
The Linker will normally group together all program sections that are writable into one virtual memory segment.

Fetchable Section

The Linker will assign a Fetchable program section to a virtual memory segment which contains only other Fetchable program sections.
To be Fetchable, all data references to the resulting segment must be self contained within the virtual memory segment. That is no statement outside of the segment can have a data reference to any part of it. As an example we may consider a section that consists of a subroutine that reads its own contents. Such a section is writable and readable and may be made Fetchable as long as no other part of the program tries to read the subroutine.

Interpreted Section

This attribute informs the Linker that the program section contains statements to be interpreted.

Debug Section

This attribute informs the Linker that the section contains only debug information which must be included in the Internal Symbol Dictionary file.

Linked Section

This attribute specifies that the section describes a virtual memory segment of a previous link step.
This attribute may not be set by the user.
The use of this section attribute allows the Linker to pass segment and symbol definitions from one link step to another.

Stack Section

This attribute specifies that the section describes a Stack memory segment.
All relocatable Stack sections are assigned to the user data Stack segment.

Assigning the Sections

The virtual memory segments are assigned to the program sections according to:

- the names and/or attributes of the sections (see the GROUP command in Chapter 3)
- the user's directives (see the ASSIGN, BLOCK DESCRIPTOR and SEGMENT commands in Chapter 3).

The logical space of the segments is divided into regions:

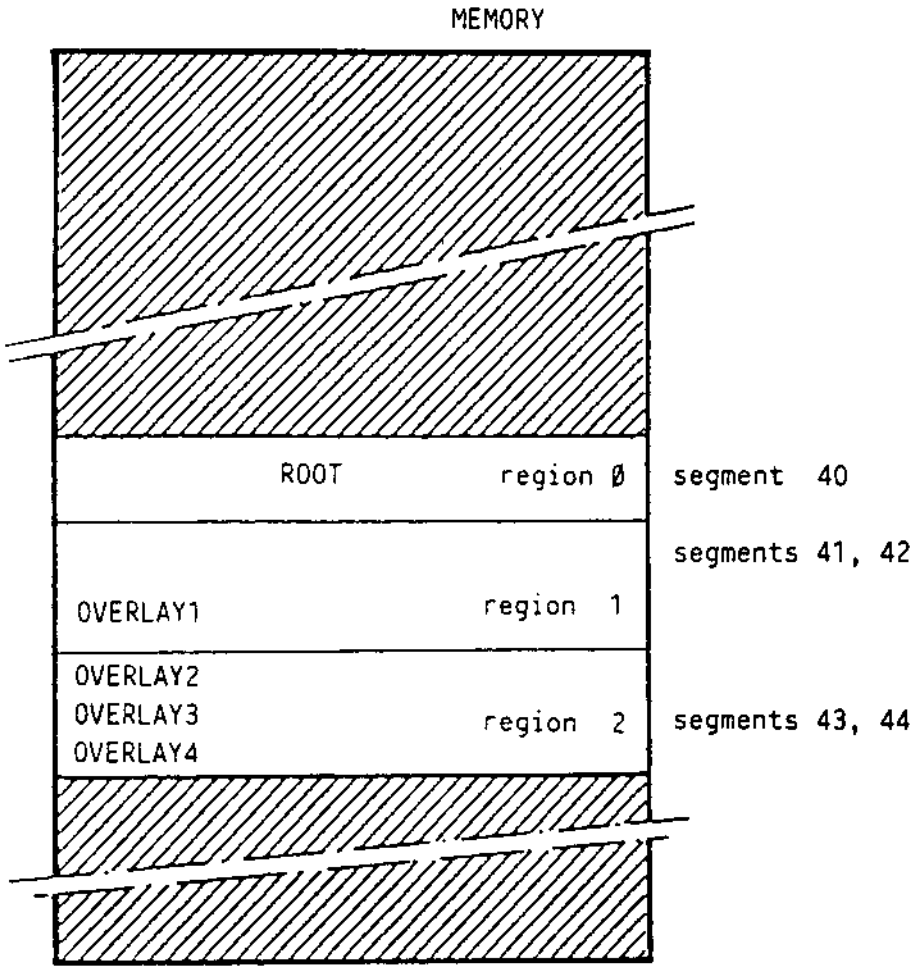
- The regions reserved for the system (one or more).
- The ROOT region, which is always present, to which the MAIN load module is assigned.
- Other regions defined by the user for building overlays.

Unless it is absolutely necessary, the user need not define the segments to be used as this is done by the Linker, according to the algorithm for the SEGDOWN/NO SEGDOWN option (see the OPTIONS command in Chapter 3).

The code, constants and sometimes data segments are usually found in the overlays. All the other segments are part of ROOT.

OVERLAY

The Linker permits the user to define an overlay structure for the virtual memory segments generated. The purpose of memory segment overlays is to enable the segments to be used more than once when no more memory area is available. The use of overlays allows memory space to be optimized by grouping code sections having similar functions, in separate overlays. Each overlay may be loaded into virtual memory if so requested by the program. In this manner the memory space is conceptually expanded.
An overlay structure in virtual memory is defined in terms of a set of regions, each of which has a virtual memory area of one or more segments allocated to it. The following figure illustrates how one or more overlays are defined within each region.



As shown in the figure there is a special region identified as the "ROOT". Within this region only the MAIN is found. "ROOT" coordinates all the overlays. The other regions defined by the user may contain several overlays.

The user defines an overlay structure by naming the modules and/or sections that make up each overlay within each region.
The segment numbers chosen for a module which is within an overlay must be within the range of segment numbers assigned to the region which contains the overlay.
MAIN and overlays are all load modules.
The set of segment numbers allocated to a region may be explicitly defined by the user or implicitly determined by the Linker.

SYMBOL RESOLUTION

The definition of external symbols allows the same data and the same code to be used by more than one module.
An external symbol is defined within a program section which contains the code or data that is associated with that symbol. It may be referred to by any object module which needs to use that code or data. Each external symbol has a type, a section, an offset and a name. Reference to a symbol must be made by specifying its name and type.
The process of symbol resolution consists of the following three stages (not necessarily carried out in this order):

- Symbol collection from input object modules. A symbol must be defined only once but may be referred to several times.

- Symbol scan of object libraries. In reality the scan is carried out only on the Symbol directory and not on the entire directory.

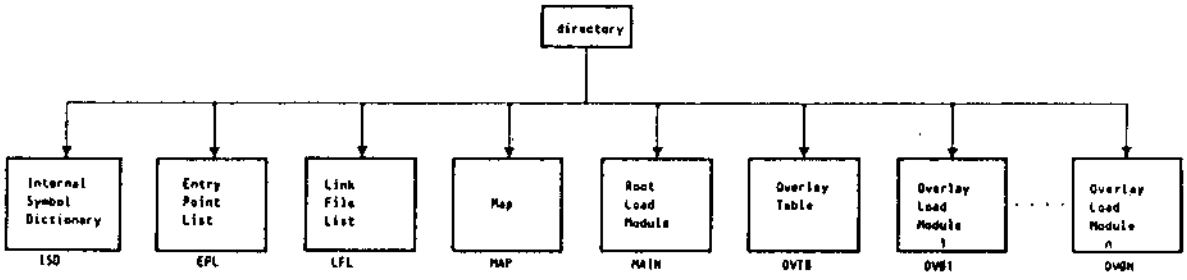
- Symbol inclusion from other Load Modules created by a previous link step. This means that an "Entry Point List" file must have been created in the load-module by a previous link step. This file defines the virtual address space that the previously linked load-module has assigned to it and specifies the external symbols that are defined within it. These symbols are included in the current link step and the virtual address space in which they reside is reserved so that the two load modules can coexist in it. An entry is made in the Link File List file, that is a part of the Load Module, to indicate that the new load module refers to symbols within the previously linked load module.

SYMBOL TYPE CHECKING

A symbol type is associated with each symbol definition and reference.
The Linker will check that all references to a symbol are compatible with the symbol type definition.

PROGRAM DIRECTORY STRUCTURE

The Program Directory is the principle output of the Linker.
Each file which is under the program directory groups a specific type of information having a distinct use within the system.
The following figure illustrates the structure of a program directory.



where:

- The Internal Symbol Dictionary file constitutes the mechanism by which the language translator communicates the symbolic description of code and data, as defined in the source program, to the symbolic debugger.
- The Entry Point List file, contains the external entry point definitions that are contained in the load modules which may be referred to by other load modules in successive link steps.
- The Link File List file contains the list of all the load modules included in the program virtual address space.
- The Map file contains all the user error messages and the maps or listings which the Linker may provide in output.
- The Root Load Module file contains the code and data segments which constitute the main program.

- The file which contains the Overlay table is automatically generated when an overlay structure is created.

- The Overlay Load Module files contain the code and data segments of the program overlays. Each overlay is stored in a separate file in the directory.

GENERATING LISTS AND MAPS

The Linker provides the following output listings:

- Module list

- Memory map

- Cross reference list

The Module list gives the name and the list of all the sections of the module, for each module. It also specifies the size and attributes of each section.
At the end of the Module list there is a part containing statistics, which gives the number of sections defined.

The Memory map is an ordered list of segments. For each segment is specified its type and the ordered list of the input sections.
For each section it gives the name, the start address, the end address and the name of the module which contains it.

The Cross-reference list gives an alphabetically ordered listing of all symbol names showing their value and the name of the module that defined them as well as all modules that refer to them. It also gives additional listings of undefined and unreferenced symbols.
At the end of the Cross-reference list there is a part containing statistics, which shows how many global symbols have been defined and how many references there are to symbols.

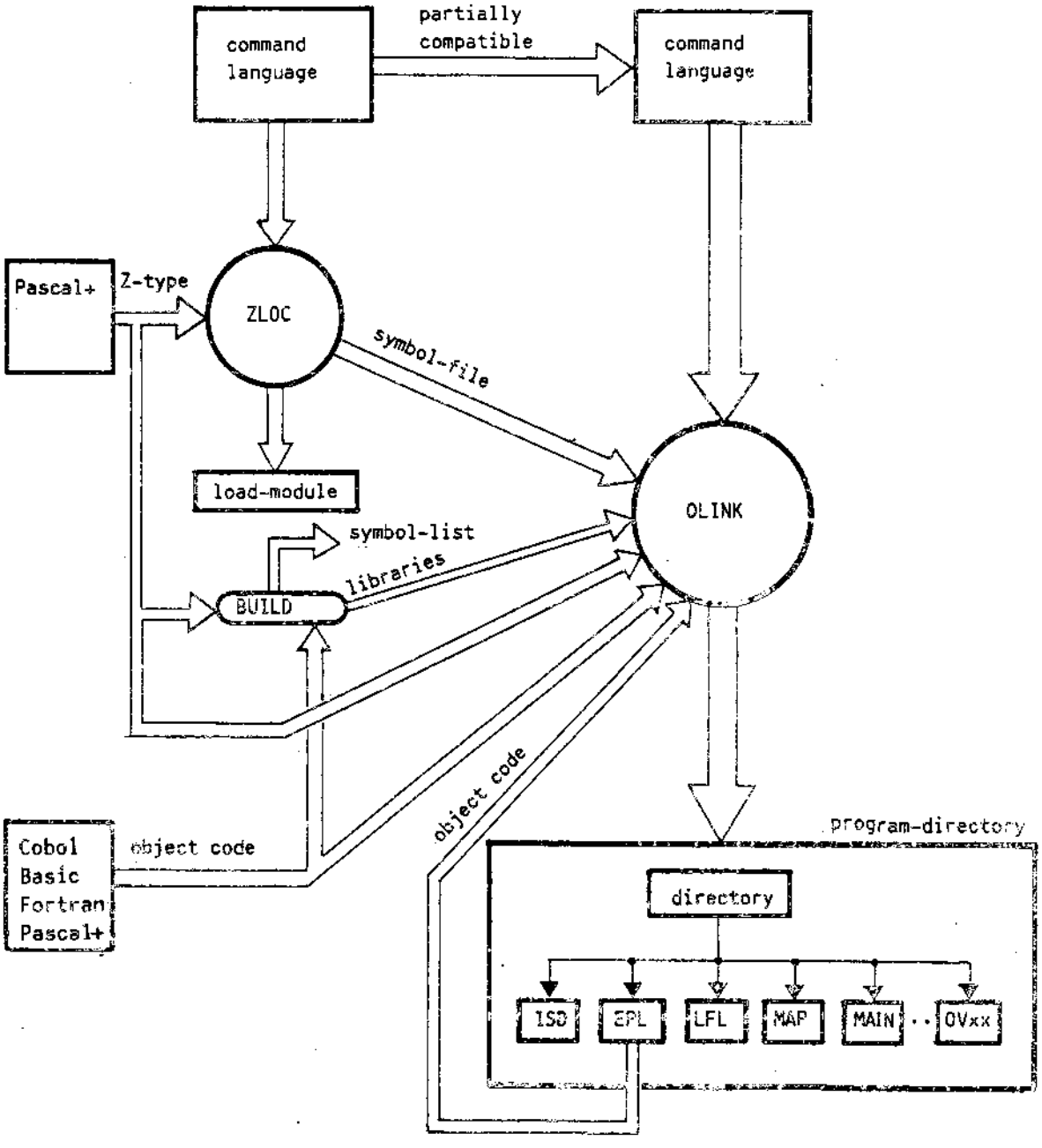
BUILDING LIBRARIES

Apart from the translator program the Linker has another program (BUILD) for the construction of files containing object module libraries. This program allows the inclusion of the single object modules, which resolve the references to undefined symbols, in the load modules during the link phase. Thus the library resolves some of the remaining undefined symbols. at program call time.

GENERAL SCHEME

The figure below shows the various interfaces which exist between OLINK and its users.

Note that OLINK cannot handle ZLOC libraries.



Where input to OLINK may be:

- The command language, partially compatible with that of the Pascal+ Linker (ZLOC).
- The symbol file produced by the ZLOC Linker.
- The libraries built via the BUILD program.
- The object modules resulting from the compilation of programs in BASIC, COBOL, FORTRAN, Pascal+.

Where OLINK output is:

- A program directory having the same format as that described in the subsection "Program directory structure".

CHARACTERISTICS AND RESTRICTIONS

When the linker receives the specified commands it carries out the following operations:

- all the "GROUPS" are processed in the order they have been input
- all the overlay section that have the same name as the file EPL are excluded
- all the SEGMENT, ASSIGN, SECTION, RETAIN, DELETE, RETAINSEC, and DELETESEC commands are processed in the order they have been input.

The following MOS types:

- RLRDCODE : code only relocated for read
- RLRWCODE : code relocated for read and write
- RLRDDATA : data only relocated for read
- RLRWDATA : data relocated for read and write

cannot be assigned to code and data output by the BASIC, COBOL, FORTRAN and Pascal+ compilers.
OLINK checks compatibility between mmu-type and mos-type, and therefore does not signal errors if the types listed above are used. In this case the program will abort at run-time.

OLINK has the following restrictions: no more than the following can be defined:

- 127 segments
- 255 overlays
- 4096 sections
- 4096 regions
- 4096 global and common symbols
- 4096 commands,
- 4096 definitions of groups
- 4096 definitions of Cobol types
- 4096 numbers of modules
- 8192 references to symbols
- 14000 names, e.g.:

  . section names, group names, symbols, file names (object files, command files and libraries), overlay names, module names and region names.

2. USING THE LINKER

The Generalised Linker is called in the Shell environment by specifying:

- the keyword OLINK
- the files to be linked
- the directives wanted.

These directives are made up of a set of commands (see Chapter 3) and can be used at three different levels:

- Using the default file witnout modifications or with varied Linker output options (see the OPTIONS command in Chapter 3).
- Using the default file and specifying a logical object aggregation (overlay).
- Defining the segments in which the program is to be loaded.

These three levels can be used in any combination.

FIRST LEVEL: DEFAULT FILE

The default file, read by the Linker, contains the following:

```
# LINKER DEFAULTS
    OPTIONS
            NO MEMORY MAP
            NO MODULE LIST
            NO XREF LIST
            NO ISD
            NO EPL
            NO TYPECHECKING
            NO SEGDOWN
            NO IGNORE
               SILENT
               REPLACE

# allocation group definitions
    GROUP MIXED       SIZE 64
                      TYPE 2 MMU 0

    GROUP HEAP        SIZE 64
                      TYPE 5 MMU 0
                      ATTR NO STACK NO EXECUTE READ WRITE
                      NAME *_[ieh]

    GROUP CODE        SIZE 64
                      TYPE 2 MMU 1
                      ATTR NO STACK EXECUTE NO READ NO WRITE
                      NAME *_[pP]

    GROUP DATA        SIZE 64
                      TYPE 5 MMU 0
                      ATTR NO STACK NO EXECUTE READ WRITE
                      NAME *_[dD]

    GROUP CONSTANT    SIZE 64
                      TYPE 4 MMU 1
                      ATTR NO STACK NO EXECUTE NO WRITE READ
                      NAME *_[kK]

    GROUP STACK       SIZE 64
                      TYPE 6 MMU 32
                      ATTR STACK
                      NAME *_[sS]

    GROUP ISD         SIZE 128
                      ATTR DEBUG

# overlay mode
    SEPARATE

# reserve system1 segments first MMU
    REGION SYSTEM1 USE  0 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 31 32 33 34 35
                       36 37 38 39 62

# reserve system2 segments second MMU
    REGION SYSTEM2 USE 64 65 66 67 68 69 70 71 72 73 74
                       75 76 77 78 79 80 81 82 83 84 85
                       86 87 88 89 90 91 92 93 94 95

# reserve for hw with only one MMU
    REGION NOMMU2 USE  96  97  98  99 100 101 102 103 104
                      105 106 107 108 109 110 111 112 113
                      114 115 116 117 118 119 120 121 122
                      123 124 125 126 127

# start with root
    ROOT
```

The default file is a Linker command file and makes the linking process very easy. It is read at the start of a program linking phase and resides in the Linker's program directory. If the user does not have any special requirements the default file only can be used.

Example 1

```
OLINK PROGRAM FATT.E OBJ1 OBJ2 LIB1 RT-Command-File
```

where:

- "OLINK" is the call for the Linker.
- "PROGRAM FATT.E" specifies the program directory.
- "OBJ1 OBJ2" are the files, and "LIB1" is the library to be linked.
- "RT-Command-File" are the libraries and commands required at run time for the language being used (see "COBOL - Program Preparation and Execution" and "Compiled BASIC - Program Preparation and Execution"). The output is:
  - a program directory, FATT.E
  - a load module, known as MAIN, under the FATT.E program directory.

If the user does not give the PROGRAM command, the Linker's default name is the name of the first module given.

Operating in this way, the Linker produces in output a program directory containing the MAIN made up of all the sections of all the object files given. The following message is displayed on the standard output if errors occur:

ERRORS —> LOOK IN THE MAP FILE

which tells the user to check the map. All the options (see the OPTIONS command in Chapter 3) are normally inactive and do not give any additional output for the load module. In the example below the specified options override those given in the default file.

Example 2

```
OLINK PROGRAM FATT.E OBJ1 OBJ2 LIB1  RT-Command-File OPTIONS NO SILENT
XREF
```

where:

- "NO SILENT" indicates that warning messages and the OLINK map are to be displayed.

- "XREF" indicates that the cross reference is wanted.

The default file is the base on which programs are built. It can be modified by the System Administrator of the machine defining a common operating mode for all the users. The commands are processed by the Linker in the same sequence in which they are entered, starting from the default file. This must always be present even if it is empty; it is processed by the Linker before the user commands.

DESCRIBING THE DEFAULT FILE

The standard Linker directives are given in the default file, and are described below.

Assigning the Options

All the options are assigned so that the least number of files possible is generated:

- No map is produced.

- Warning messages are not given.

- The EPL and ISD files are suppressed.

- The segments to be assigned to the programs are chosen, starting from the highest numbers.

Defining the Allocation Groups

The default file contains groups definitions (see the GROUP command in Chapter 3) so that separate segments contain:

- Code (see "MODIFYING THE DEFAULT FILE")
- Constants (see "MODIFYING THE DEFAULT FILE")
- Read/write data
- Stack
- Heap

These are the main types of segments making up an executable program. A group is also defined for debugging the ISD file (see the OPTIONS command in Chapter 3). The list of groups is scanned as each section is input. The Linker checks that the section's name or attributes are compatible with the group. If at least one of these tests is positive, the section is allocated to that group. If no group is found for a particular section, this is allocated to the first group defined starting from the default file.

Defining the Segments which are not to be used

The indicated segments are reserved for the system. This indication means that the logical address space is inaccessible during the current linking phase (see the USE command in Chapter 3).

Root

This command assigns the current region to the program's MAIN.

MODIFYING THE DEFAULT FILE

As the logical address space is limited, the default file sometimes has to be modified to group sections which would normally be separated. One way of doing this is to mix code and constants in the same group.

Example:

```
# LINKER DEFAULTS
    OPTIONS
            NO MEMORY MAP
            NO MODULE LIST
            NO XREF LIST
            NO ISD
            NO EPL
            NO TYPECHECKING
            NO SEGDOWN
            NO IGNORE
               SILENT
               REPLACE

# allocation group definitions
    GROUP MIXED       SIZE 64
                      TYPE 2 MMU 0

    GROUP HEAP        SIZE 64
                      TYPE 5 MMU 0
                      ATTR NO STACK NO EXECUTE READ WRITE
                      NAME *_[ieh]

    GROUP COD_CONST   SIZE 64
                      TYPE 2 MMU 1
                      ATTR NO STACK NO WRITE
                      NAME *_[pPkK]

    GROUP DATA        SIZE 64
                      TYPE 5 MMU 0
                      ATTR NO STACK NO EXECUTE READ WRITE
                      NAME *_[dD]


    GROUP STACK       SIZE 64
                      TYPE 6 MMU 32
                      ATTR STACK
                      NAME *_[sS]

    GROUP ISD         SIZE 128
                      ATTR DEBUG

# overlay mode
    SEPARATE

# reserve system1 segments first MMU
    REGION SYSTEM1 USE  0 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 31 32 33 34 35
                        36 37 38 39 63

# reserve system2 segments second MMU
    REGION SYSTEM2 USE 64 65 66 67 68 69 70 71 72 73 74
                       75 76 77 78 79 80 81 82 83 84 85
                       86 87 88 89 90 91 92 93 94 95

# reserve for hw with only one MMU
    REGION NOMMU2 USE  96  97  98  99 100 101 102 103 104
                      105 106 107 108 109 110 111 112 113
                      114 115 116 117 118 119 120 121 122
                      123 124 125 126 127

# start with root
    ROOT
```

SECOND LEVEL: LOGICAL OBJECT AGGREGATION

The Linker allows the user to logically group output objects, without having to define the program segment by segment. This operating mode is based on the following commands:

- REGION
- GROUP
- OVERLAY

Example 3

```
OLINK PROGRAM FATT.E  OBJ1 REGION ALFA OVERLAY OV1 OBJ2 OVERLAY OV2  OBJ3
LIB1 RT-Command-File OPTIONS NO SILENT
```

where:

- "OLINK" is the call to the Linker.
- "PROGRAM FATT.E" specifies the program directory.
- "OBJ1" indicates that it must be placed in Root.
- "REGION ALFA" defines the ALFA region in which the OV1 overlay, consisting of OBJ2, and the OV2 overlay, consisting of OBJ3 and LIB1, are found.
- "RT-Command-File" are the libraries and commands required at run time for the language being used (see "COBOL - Program Preparation and Execution" and "Compiled BASIC - Program Preparation and Execution").
- "OPTIONS NO SILENT" indicates the desired option.

The REGION command (see Chapter 3) divides the available virtual address space into logical partitions.

The GROUP command (see Chapter 3) establishes how the input objects are to be grouped. Input objects are the sections of the files (compilation units) to be linked.

The OVERLAY command (see Chapter 3) defines a program overlay on the current region. Each overlay is made up of a separate load module called OVxx. The main load module and the eventual overlays are placed in the program directory.

USING THE COMMAND FILES

The user can indicate to the Linker which commands he wants to use by:

- creating a command file in EDITOR which contains the required directives
- specifying the commands in the Linker's call, following the Shell environment's conventions.

The following two examples illustrate this and are equivalent.

```
OLINK PROGRAM FATT.E OPTIONS NO SILENT XREF MEMORY MAP OBJ1 OBJ2
RT-Command-File
```

```
OLINK cmdfile OBJ1 OBJ2 RT-Command-File
```

where

- "cmdfile" contains the directive:

  PROGRAM FATT.E OPTIONS NO SILENT XREF MEMORY MAP

The command files can be nested, allowing them to refer to each other. There can be up to seven nesting levels. This is shown in the map in the section "DIAGNOSTICS, Command Language Interpretation".

THIRD LEVEL: SEGMENT DEFINITION

If there are any special requirements, and the segment and offset of each section have to be established, the following commands can be used (see Chapter 3):

- BLOCK DESCRIPTOR
- SEGMENT
- SECTION
- TYPE
- ATTRIBUTE
- ASSIGN

Some examples of using these directives are given below.

```
        .
        .
        .
ATTRIBUTE 1
TYPE 2
<55>%0000 *_[pP]
        .
        .
        .
```

```
        .
        .
        .
SEGMENT 55
    MMU  1  TYPE 2
    OFFSET  %0000 *_[pP]
ENDSEGMENT
        .
        .
        .
```

Segment 55 is created with executable code (TYPE 2, MMU 1) in the two examples above, with the directives specified in a command file. An alternative to these methods is:

```
        .
        .
        .
SEGMENT 55 IS CODE
OFFSET  %0000 *_[pP]
ENDSEGMENT
        .
        .
        .
```

Segment 55 can be defined in the way described above as the CODE group with TYPE 2 and MMU 1 is defined in the default file.

Example:

```
        .
        .
        .
ATTRIBUTE 1
TYPE 2
<55> %0000 *_[pP]
     %8000 *_[kK]
        .
        .
        .
```

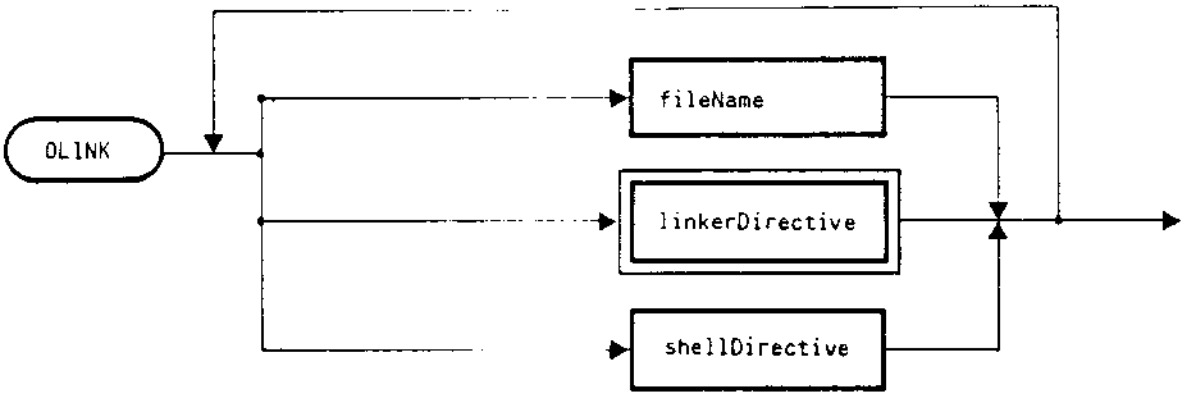
```
        .
        .
        .
SEGMENT 55
    MMU  1  TYPE 2
    OFFSET  %0000 *_[pP]
    OFFSET  %8000 *_[kK]
ENDSEGMENT
        .
        .
        .
```

If these directives are given in a command file a segment is created with read only code and data.

Note that a segment cannot be redefined.

CALLING THE LINKER

The Linker's call must have the following syntax:

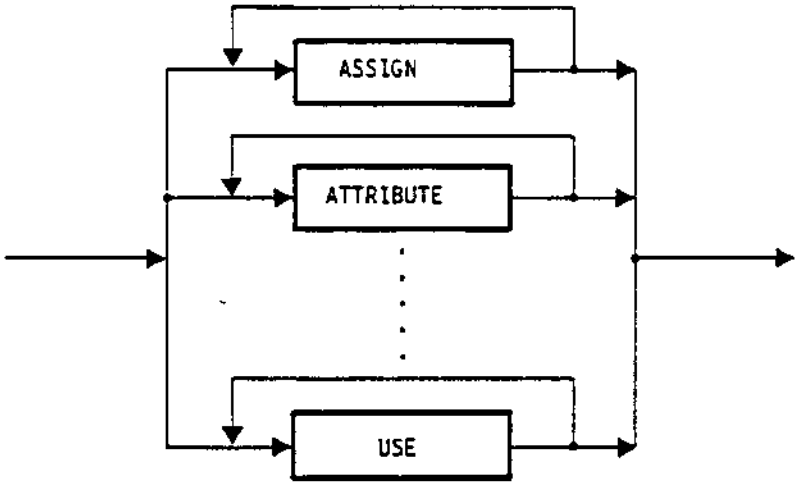


where:

- "fileName" is the name of an OLINK input file which may be one or more of the following:
  - an object file
  - a library object file
  - a load module
  - a Linker command file
  - a symbols file.

The file is identified by its pathname, which follows the same conventions as for the SHELL environment. The Linker can identify the type of each input file and so process it correctly. The Linker may be called more than once under any one working directory.

- "linkerDirective" represents the commands for the Linker as described in the following chapter in reference form. The call syntax is:



- "shellDirective" specifies the standard output files to be used by the Linker. These commands are handled by Shell but are invisible to the Linker. They are usually output redirection or background commands.

Notes

File names and Linker commands are processed sequentially in the same order in which they were entered.

In the case of an object file the file contents are included in the program in the link phase.

In the case of a library object file the library is scanned to resolve all the external references which the Linker has come across in the previously included modules.
When object modules are included, either via an object file or via a library scan they will form part of the region/overlay under construction.

In the case of a load module, the Linker includes the Entry-Point List symbol file in the program. This ensures that the virtual memory segments used by the load module are reserved and the external callable symbols are defined. The symbol file is compatible with the ZLOC Linker and may be produced directly by it.

In the case of a Linker command file, it reads all the commands contained in the file and replaces the command file names with the commands they contain.

CHAINING RUN TIME COMMAND FILES

When the Linker is called the user must specify one of the following command files:

- /IPL/DPC/COB_RTS/LIB/RTLINK if the user is COBOL
- /IPL/DPC/BAS_RTS/link.cmd if the user is BASIC

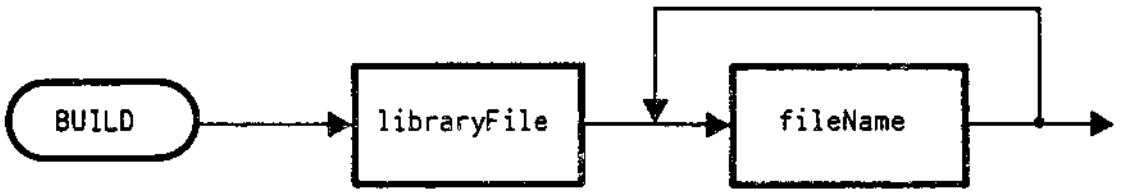
These files contain the commands necessary for loading the following system components: VISA, Line Manager, PGU, etc.
In the Linker symbol resolution phase the Linker will see if any of the symbols which are not resolved, are defined in one of the above components. If so, the component will be linked and loaded into memory.

BUILD

This is a program which allows the user to build libraries.

This program scans the object file in input and groups the names of the symbols and modules in a listing (LibraryFile.L). This is divided into two parts, the first contains an unordered listing of all the symbols contained in each module, the second part contains the ordered listing of all the symbols contained within all the modules.

The program input parameters must not exceed 140 characters.
The syntax of the program call is:



where:

- "libraryFile" identifies the library to be created.
- "fileName" identifies the object file which is to belong to the library

MEMORY ALLOCATION

The memory is made up of 127 segments, and only part of it is available to the user.

The default file specifies that segments 0 to 39 and segment 62 may not be accessed by the user, if there is one MMU.
If there are two MMUs, the system allocates segments 64 to 95, leaving segments 96 to 127 free for the user. For information on the free segments see MOS - Programmer Guide .
There are other segments which the user may not access in each operating environment.
The user can see how the memory has been allocated by listing the contents of the "MAP" file, which is in the program-directory.

START/END OF LINKING PHASE

When linking starts the following message is displayed:

*** OLINK 7.X ***

When linking terminates and no errors have occurred, the program goes back to the prompt. Otherwise the following message is displayed:

ERRORS --> LOOK IN THE MAP FILE

The value of the %STATUS variable may be tested from the Shell environment, to see if linking terminated correctly:

- %STATUS = 0 : correct end
- %STATUS = 1 : non-fatal error (warning)
- %STATUS = 2 : fatal error (abort)

If the Linker was unable to complete the linking phase correctly, or there were in any case , the error and/or warning messages are stored in the MAP file, which is in the program-directory.
The Linker then displays the following message:

ERRORS --> LOOK IN THE MAP FILE

This is the Linker's way of passing information to the users, who can read the section of the MAP file containing the error messages, provided that he had previously requested its creation.

In certain cases of abnormal termination of the linker, the following files may remain open in the current directory: VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn. "nnnnnnnn" are eight hexadecimal characters repressnting the process number.

3. COMMAND LANGUAGE

This chapter describes the Linker command language, giving the main characteristics and a description of all the commands available.

LANGUAGE CHARACTERISTICS

The Linker command language allows the user to:

- Specify which modules are to be included in the program
- Specify which libraries and load modules to search for or include
- Specify the program overlay structure
- Specify the desired output (maps, listings etc.).
- Control the generation of the Internal Symbol Dictionary
- Control the generation of the Entry Point List file
- Modify the attributes of the sections which make up the program
- Assigns an address to a section within a segment.

The Linker command language is heavily dependent on the definition of the overlay structure and of the type of memory allocation adopted.
A summary of both is given below.

Regions and Overlays

A region is an area of segments.
One or more overlays may be assigned to a region. Each overlay is made up of at least one module. A module is assigned to the current region/overlay pair via a Linker command. Successive commands may assign all or part of the program sections within a module to other region/overlay pairs. All programs contain one special region known as "root" which contains just one overlay. The root region is never unloaded from the program virtual address space. This means that the data segments contained in it cannot be reinitialised and the code segments may always be referred to directly.

Allocation Groups

The Linker command language defines all memory allocation in terms of allocation groups. Each allocation group has a group name, a name pattern, a list of attributes, a maximum size, a list of MMU attributes and a type. The name pattern and attribute list are used to determine which program sections are part of the allocation group.
The Linker builds one or more virtual memory segments, for each allocation group, within each region/overlay pair that contains one or more program sections.

ZLOC COMPATIBILITY

The OLINK command language has been extended to allow a certain degree of compatibility between the ZLOC and OLINK directives. Some of these OLINK commands are now expressed in two different alternative forms:

- ZLOC type
- OLINK type

The SEGMENT (OLINK type) and BLOCK-DESCRIPTOR with TYPE and ATTRIBUTE (ZLOC type) commands create the same directive.

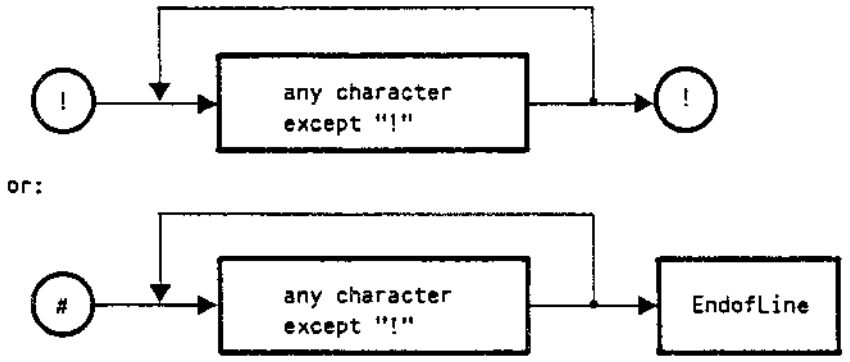
COMMANDS

All the commands, with the exception of one, start with a keyword, which may be followed by parameters and another keyword. One of the commands for the allocation of segments is an exception to this rule. To distinguish this command from others we have called it "BLOCK-DESCRIPTOR" for reference purpose.

Keywords may be entered in upper case or lower case letters. Information on how to insert comments between one command and another and the reference of all commands which have been ordered alphabetically follows.

Comments

Comments are character sequences which are not Linker commands.
The following diagram illustrates the two forms which a comment may take.



In the first case the comment delimiter is "!", whereas in the second case the comment is delimited by "#" to the left and by the end of the line to the right. A comment can not be inserted within a linker directive.

FUNCTIONAL GROUPS

The command language is structured in a manner which allows the commands to be divided and subdivided into the following functional groups.

Commands which Define Input

The Linker is able to define the file type (command file, symbol file, load module file,object file,object library file) of a file specified within one of its commands. The user may request the Linker to read and interpret the identified file by using one of the following commands:

- COMMAND
- FILE
- INCLUDE
- INPUT
- INSYM

Commands which Define Output

The Linker will create as many files as requested by the user. The majority of these files are created under the program directory. The user may produce or suppress output files by using the following commands:

For maps or listings:

- MAP
- NOWARNINGS
- OPTIONS [no] ignore
  [no] isd
  [no] memory map
  [no] module list
  [no] replace
  [no] segdown
  [no] silent
  [no] typechecking
  [no] xref list
- QUIET

For the Entry-Point List file or the symbol file:

- DELETE
- DELETESEC (*)
- OPTIONS [no] epl
- RETAIN
- RETAINSEC (*)
- SYMBOL

(*) **Note:** the commands "DELETESEC" and "RETAINSEC" are typical of the COBOL Dynamic Link.
For further details see COBOL - Program Preparation and Execution

For load module generation:

- OUTPUT / PROGRAM

Commands for the Management of Load Modules

The user may manage the load modules via the appropriate commands.

To define and manage overlays:

- ASSIGN
- COMBINE
- HONOR
- OVERLAY
- SEPARATE

To define regions:

- DATASTACK
- REGION
- ROOT
- USE

To allocate memory:

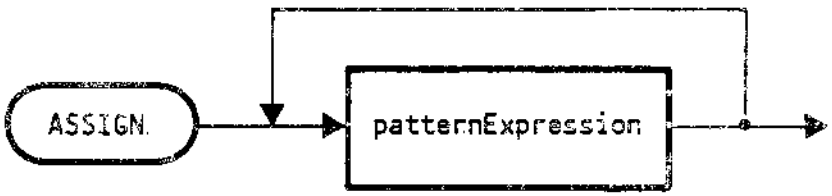
- ATTRIBUTES
- BLOCK-DESCRIPTOR
- GROUP
- SECTION
- SEGMENT
- STACKBASE
- TYPE

To assign a message or an entry point to the main load modules:

- ENTRY / ENTRYPOINT
- MESSAGE
- OPTIONS [no] segdown

ASSIGN

Assigns the specified sections to the current region/overlay. Particular module sections may be allocated to regions other than those to which the remaining module sections were allocated.
"ASSIGN" commands are processed after the Linker has read all the object modules in the same order in which they were entered.
The command syntax is:



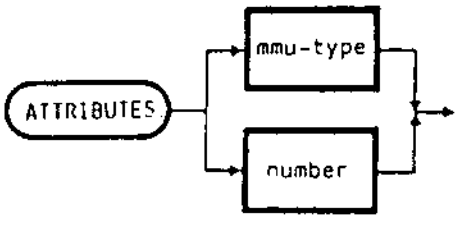
where:

- "PatternExpression" is an expression used to identify one or more sections. The characters it contains assume the following meaning:

  * : identifies any character of any type, that is any name
  ? : identifies any single character
  [abks] : identifies one of the characters specified within the square brackets, in this case a,b,k or s.

ATTRIBUTES

Assigns an attribute to a segment which has been created and establishes the type of hardware protection of the segment. Each segment created in the executable output file has an attribute and a type. The default value of the byte which represents the segment attribute is 0.
There is no limit to the amount of times that this command may recur.
This command is compatible with the ZLOC Linker.
It's syntax is:



where:

- "mmuType" specifies one of the following attributes to be assigned to the segment:

```
RDWRT    : segment which may be written to, read and executed
RDONLY   : segment which may be read and executed
XQTONLY  : segment which may only be executed
STCKATTR: segment with stack function
```

- "number" represents the attribute type code which may be used as an alternative to "mmuType":

```
 0 :RDWRT
 1 :RDONLY
 8 :XQTONLY
32 :STCKATTR
```

BLOCK-DESCRIPTOR

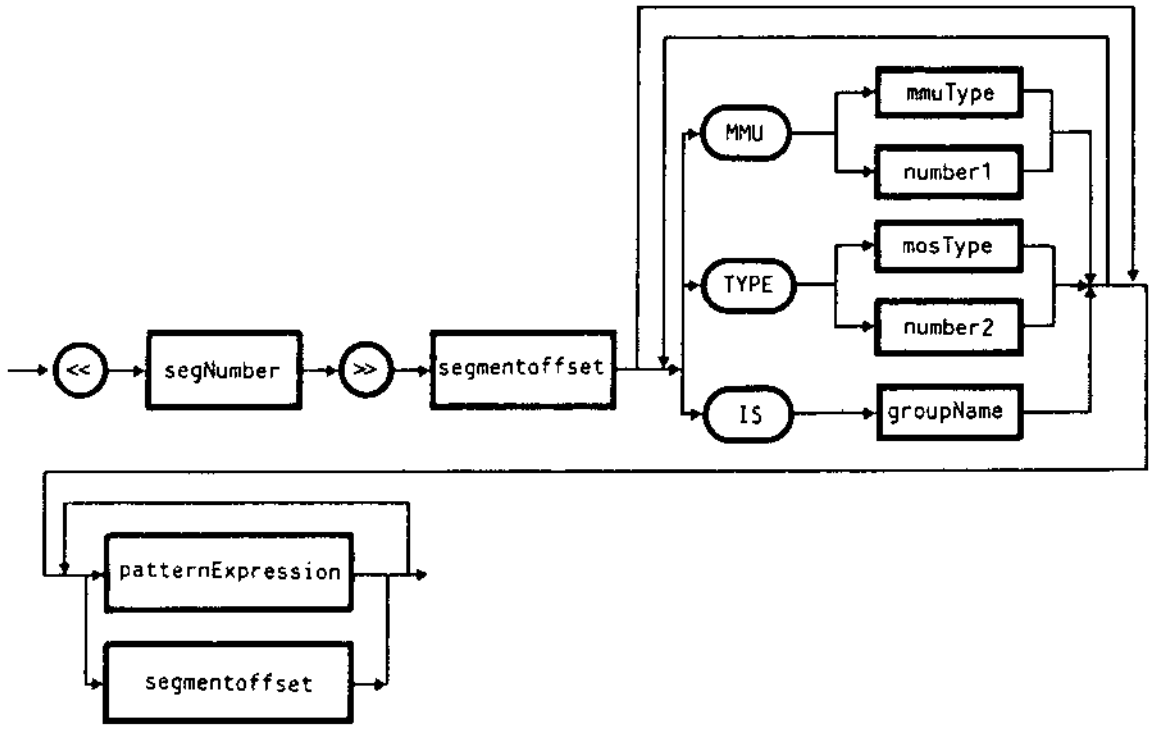
This command allocates a segment (see the SEGMENT command). Segments may be defined only once.
For example the following definition is in error:

```
<<43>>%0000 *_[rR]
<<43>>%B000 raffa
```

whereas it would be correct to define the segment thus:

```
<<43>>%0000 *_[rR]
      %B000 raffa
```

This command is compatible with the ZLOC Linker.
The command syntax is:



where:

- "segNumber" specifies the segment number and must be in the range 0 to 63. No arithmetic expressions are permitted within "segNumber".

- "segmentoffset" is a number which specifies the segment offset. No arithmetic expressions are permitted within "segmentoffset".

- "number1" represents the "mmuType" (RDWRT, RDONLY, XQTONLY, STCKATTR) code (0,1,8,32) which specifies one of the segment attributes for the segment:

```
  0 : RDWRT     - segment which may be written to, read and
                  executed
  1 : RDONLY    - segment which may be read and executed
  8 : XQTONLY   - segment which may only be executed
 32 : STCKATTR  - segment with stack function
```

- "number2" represents the "mosType" (XQTCODE, RDCODE, RWCODE, RDDATA, RWDATA, STACK) code (1,2,3,4,5,6) which specifies one on the following segment types:

```
  1 : XQTCODE  - code for execution only
  2 : RDCODE   - code for reading only
  3 : RWCODE   - code for reading and writing
  4 : RDDATA   - code for reading only
  5 : RWDATA   - data which may be written or read
  6 : STACK    - stack segment
```

- "groupName" specifies the section allocation group name.

- "patternExpression" is an expression used to identify one or more sections. The characters within it have the following meaning:

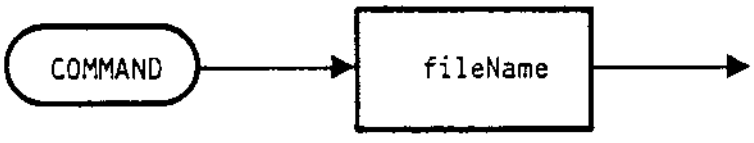
```
    * : identifies any number of characters of any type, that is
        any name
    ? : identifies any single character
[abks] : identifies one of the characters specified within square
         brackets in this case a,b,k or s.
```

COMBINE

Indicates that the overlay structure defined in the object module is to be ignored. Furthermore all the code segments are grouped into a single virtual memory segment.
The command has no parameters.

COMMAND

Used to give a set of commands, contained in the specified file, to the Linker.
The specified file in its turn may contain "COMMAND" nested up to 7 levels.
This command is compatible with the ZLOC Linker.
The command syntax is:

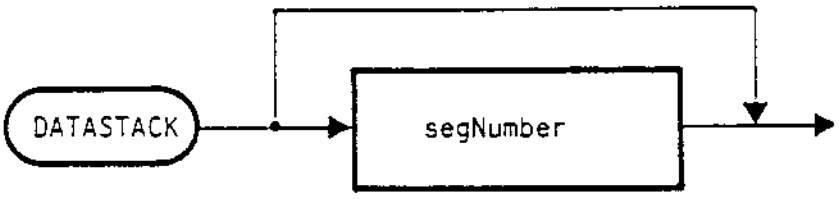


where:

- "fileName" is the name of the command file.
  The file is identified by its pathname, which follows the same conventions adopted for the Shell environment.

DATASTACK

Makes the stack region the current region. The stack region has exactly one overlay: the user-data stack. The user may specify the virtual segment number to be assigned to the user-data stack.
The command syntax is:



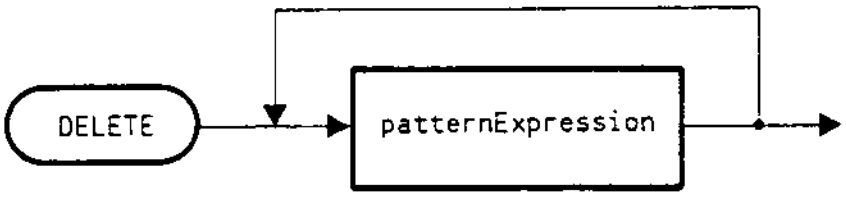
where:

- "segNumber" declares which segment is to be used for the stack, it may be in the range 0 to 63

DELETE

Specifies the symbols which are to be deleted from the Entry Point List of the load-module. It is processed after all the object modules for the linking operation have been read.
The DELETE and RETAIN commands (see the RETAIN command) are processed sequentially in the order they are given.

The command syntax is:



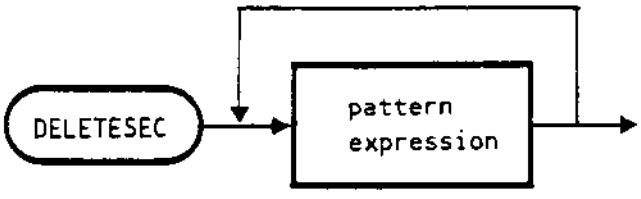
where:

- "patternExpression" is an expression used to identify the symbols which are to be deleted. The characters it contains have the following meaning:

  * : identifies any number of characters of any type, that is any name
  ? : identifies any single character
  [abks]: identifies one of the characters specified in the square brackets. In this case a,b,k, or s.

DELETESEC

This command specifies the sections with the OVERLAY attribute that must be deleted from the Entry-Point List of the load-module. It is processed after reading all the object modules to be linked. The commands DELETESEC and RETAINSEC (see the command RETAINSEC) are processed sequentially in the order they are given.
The syntax of the command is as follows:

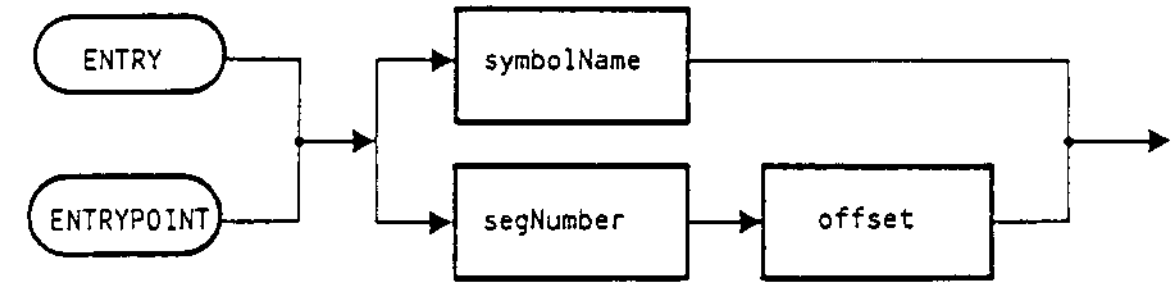


where:

pattern-expression is defined as above.

ENTRY / ENTRYPOINT

Provides a numeric value or the name of a global symbol which is the entry point of the load module.
The command may be entered only once.
It is compatible with the ZLOC Linker.
The command syntax is:



where:

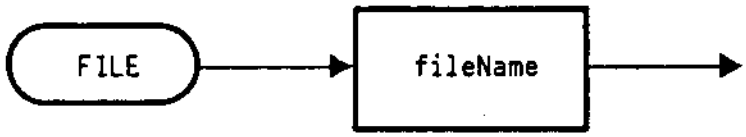
- "symbolName" specifies the global symbol name.
- "segNumber" may assume a value in the range 0 to 63. Together with the "offset" it indicates the program entry point.

Notes

- If this command is present, the name of the specified symbol or an address e.g. <segment number, offset>, is used as an entry point, independently of any definition existing within the object module.
- If it is not present, the entry point is the first that is defined in the object modules which are scanned.
- If there is no entry point declared in the object modules, the Linker establishes the start of the first code section of the first object module read as the entry point. If there is no code section, the entry point will be the start of the first section (of any type) of the first object module read.

FILE

Specifies that the parameter that follows is to be interpreted as a file name. The command is compatible with the ZLOC Linker. The command syntax is:



where:

- "filename" may indicate one of the following file types:

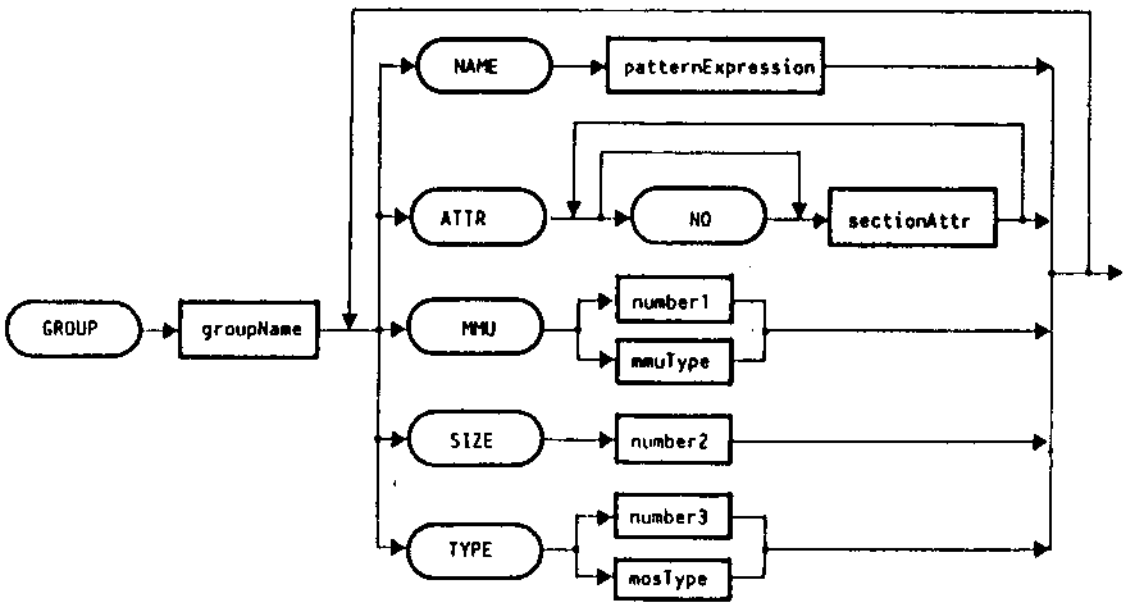
  an object file
  a library object file
  a load module file
  a symbol file
  a Linker command file

  The file is identified by its pathname, which follows the same conventions as those adopted by the Shell environment.

GROUP

Defines the sections which are to be contained in one or more segments. These sections constitute an allocation group. All the allocation groups are kept in a list within the Linker. At creation time of a new allocation group a new entry point is created at the end of the list. Ordered scanning of the list is carried out when it is necessary to identify the group to which a section belongs.
All the object modules have to be read before sections may be assigned to the various groups. The algorithm for this is:
the Linker first attempts to compare the section by using its attributes. If this fails it attempts to use its name. If this also fails an error signal is sent out and the section is assigned to the first group defined in the command list. Therefore, the first group must always be of the type defined by the default file.

The command syntax is:



where:

- "groupName" indicates the name of the group which is to contain the sections.

- "patternExpression" is an expression used to identify one or more sections which belong to the group. The characters it contains have the following meaning:

```
        * : identifies any number of characters of any type, that is
            any name
        ? : identifies any one single character
    [abks]: identifies one of the characters specified in square
            brackets in this case a,b,k or s.
```

- "sectionAttr" provides the section attributes. These may be:

```
EXECUTE
READ
WRITE
ABSOLUTE
RELOCATE
PRIVATE
SHARE
FETCH
OVERLAY
CONCATENATE
INTERPRET
LINKED
STACK
DEBUG
```

  Attributes which are not specified are not relevant to the section.

- "number1" represents the "mmuType" (RDWRT,RDONLY,XQTONLY,STCKATTR) code (0,1,8,32) which specifies one of the following segment attributes:

```
 0 : RDWRT    - segment which may be written to read and
                executed
 1 : RDONLY   - segment which may be read and executed
 8 : XQTONLY  - segment which may be only executed
32 : STCKATTR - segment with stack function
```

- "number2" specifies the maximum segment size in k-bytes.

- "number3" represents the "mosType" (XQTCODE,RDCODE,RWCODE,RDDATA, RWDATA, STACK) code (1,2,3,4,5,6) which has one of the following values:

```
1 : XQTCODE  - code used for execution only
2 : RDCODE   - code used for reading only
3 : RWCODE   - code used for read/write operations
4 : RDDATA   - data which may only be read
5 : RWDATA   - data which may written and read
6 : STACK    - stack segment
```

When assigning values to "number1" and "number3", the following limitations on compatibility must be respected:

number1	number3
0	3, 5,
1	2, 4,
8	1
32	6

HONOR

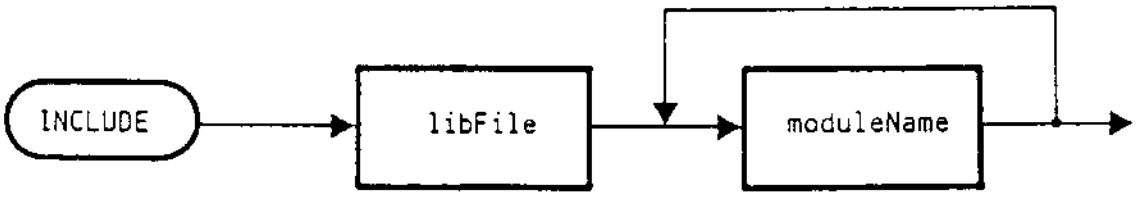
Indicates that the overlay structure ,defined by the object module, must be built using the current overlays.

This command has no parameters.

INCLUDE

Allows the modules specified in the command to be included in the current region/overlay. The modules are drawn from the indicated library object file created by the BUILD program. Library modules may thus be included even if the Linker has not come across references to the symbols defined in these modules in the modules already read.

The command syntax is:

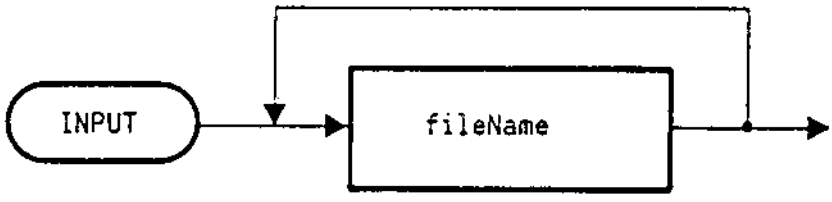


where:

- "libFile" is the library file path name.
- "moduleName" is the name of the module to be included.

INPUT

Indicates the object files which contain all the code sections which the Linker must link together.
It may recur in the command list as many times as necessary.
This command is compatible with the ZLOC Linker.
The command syntax is:

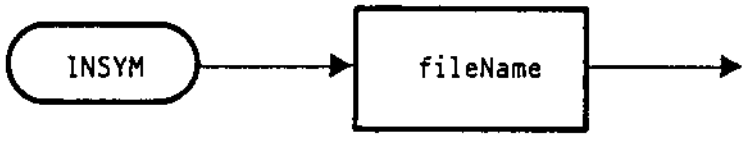


where:

- "fileName" identifies an object file.
  The file is identified by its pathname, which follows the same conventions as those adopted in the Shell environment.

INSYM

Imports symbols from another link operation.
There is no limit to the amount of times that it may recur.
The symbol file may not be only partially read.
The command syntax is:



where:

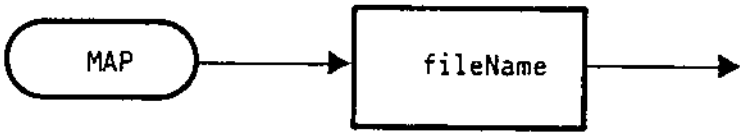
- "fileName" identifies the symbol file. The file can be produced by OLINK or by ZLOC (see the General Scheme in Chapter 1). The file is identified by its pathname, which follows the same conventions as the Shell environment.

MAP

This command executes the following three maps:

- Cross Reference List
- Memory Map
- Module List.

The file specified in the command contains the maps. Parts of maps are added to this file according to the options given (see the OPTIONS command).
The command syntax is:

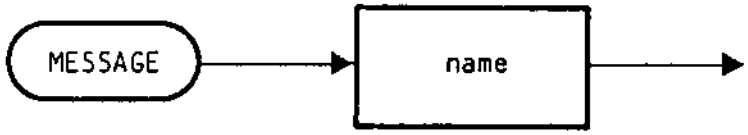


where:

- "fileName" is the pathname of the map file.
  The file is identified by its pathname which follows the same conventions as the Shell environment.

MESSAGE

Provides the text to be included in the message part of the executable file header. It may be used only once.
The command syntax is:



where:

- "name" is a string containing a message, this must not contain any blanks.

Note

If this command is not used, the Linker will include a null message in the executable file.

NOWARNINGS

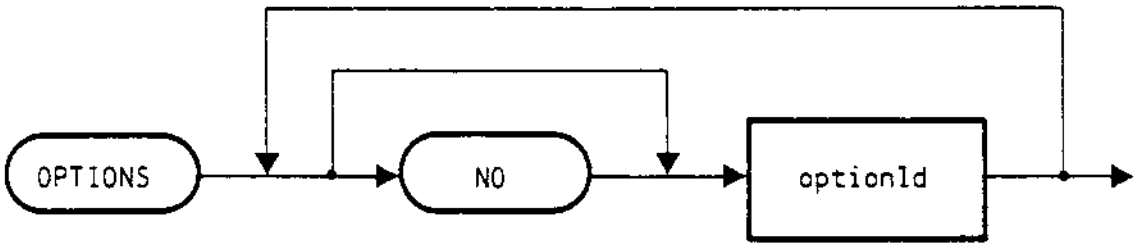
Suppresses all the warning messages.
The command has no parameters and it is compatible with the ZLOC Linker.

OPTIMIZE

This command is compatible with the ZLOC Linker and has no effect for OLINK, in fact the optimisation of data section numbers is automatic.

OPTIONS

Specifies the output options produced by the Linker. The options used by the Linker are those activated at the end of the commands. This command can be repeated.
The command syntax is:



where:

- "optionId" specifies the output options which may be:

EPL: the Entry Point List file which forms part of the load module used by the successive link steps. The EPL file is an object file which only defines sections and symbols. Either all the sections or those for the EPL file have the "LINKED" attribute. The following is contained in the EPL file:

- the symbols selected by the command sequence RETAIN/DELETE (see the RETAIN/DELETE commands) to pass them to another linking step

- the sections selected by the command sequence RETAINSEC/DELETESEC (see the RETAINSEC/DELETESEC commands) to pass them to another linking step

- the definition of the segments, as absolute sections, that are used in the current linking step and are consequently not to be used in the next one.

Correspondence between the sections defined in the EPL file (which all have the OVERLAY attribute) ad those defined in the other object files which are part of the same link, is done on the basis of the original types, not those that result from execution of the link commands. This means that correspondence is obtained before execution of all the commands to the link step, and therefore directives such as SECTION, SEGMENT, ASSIGN (see the corresponding commands) do not change section allocation. If the commands RETAIN and/or RETAINSEC are not specified, only the segments allocated as absolute sections are specified in the EPL file. Any later link step that uses this EPL file in input, will not allocate anything in the segments that have been defined in it.

IGNORE: enables the link to match names without distinguishing between upper and lower case characters.

ISD: the Internal Symbol Dictionary used by the symbolic debugger.

MEMORY or
MEMORY Map: the memory Map.

MODULE or
MODULE LIST: the Module List.

REPLACE: the Linker replaces the program directory if this already exists. If "NO REPLACE" is specified the following message is produced:
"PROGRAM DIRECTORY ALREADY EXISTS"
"INPUT A NEW PROGRAM NAME:"
This message will remain displayed until a new program name is entered.

SEGDOWN: the search for free segments starts from segment number 63 to 0.

SILENT: the Linker may operate in transparent mode. It does not give either warning messages or the first section of the map (DIAGNOSTICS), made up of:

- command language interpretation
- memory allocation
- load module generation

The effect is the same as specifying QUIET and NOWARNINGS.

TYPECHECKING: the Linker may carry out typechecking.

XREF or
XREF LIST: the Cross Reference list.

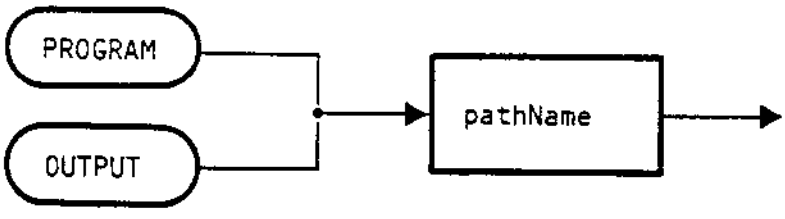
Note

The table below indicates the output generated by certain options:

options	output
memory map	MEMORY MAP SEGMENT SUMMARY
xref list	CROSS REFERENCE LIST UNDEFINED SYMBOLS
module list	MODULE LIST
silent	DIAGNOSTICS (suppresses warning/error messages).

OUTPUT / PROGRAM

Specifies the file in which the executable file produced by the Linker is to be loaded.
If this command has not been used the load module is loaded under the working directory which is active at Linker call time.
This command is compatible with the ZLOC Linker.
The command syntax is:

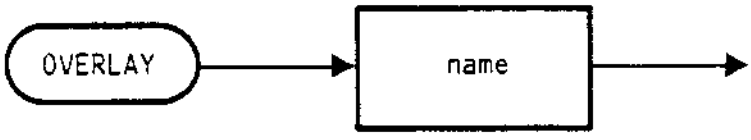


where:

- "Pathname" is a load module file, it follows the same conventions adopted by the Shell environment.

OVERLAY

Creates region/overlay pairs. The region is the current one and the overlay is that specified in the command. If the current region is the root or stack region, this command may not be used.
The command syntax is:



where :

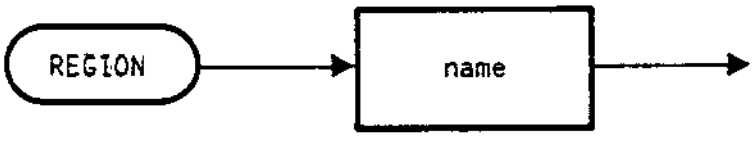
- "pathName" is the name of the load module file, it follows the same conventions adopted for the Shell environment.

QUIET

Suspends the Linker output which is normally displayed on screen.
The "fatal error" messages are however not effected by this command and will be displayed if necessary.
This command has no parameters and is compatible with the ZLOC Linker.

REGION

The specified region becomes the current one with this command (see the USE command).
The command syntax is:

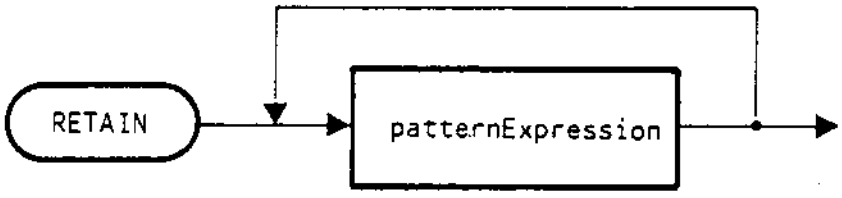


where:

- "name" is the string which identifies the region.

RETAIN

Specifies the symbols which are to be held in the Entry Point List of the load-module. The RETAIN command is performed after all the object modules for the link operation have been read. The commands RETAIN/DELETE (see the DELETE command) are performed sequentially in the order they are given.
It must be used with the EPL option (See OPTIONS EPL), otherwise it has no effect. The command syntax is:



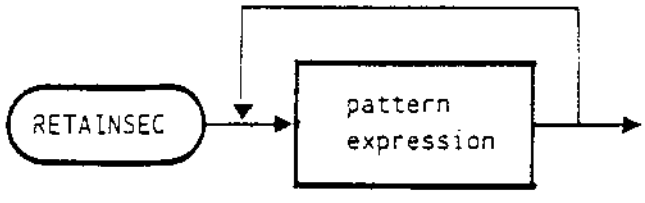
where:

- "patternExpression" is an expression used to identify symbols which are to be held in the file. The characters it contains assume the following values:

  * : identifies zero of more characters of any type, that is any name
  ? : identifies any single character
  [abks]: identifies one of the characters specified in the square brackets, in this case a,b,k or s.

RETAINSEC

This specifies the sections (which all have the OVERLAY attribute) which must be stored in the Entry Point List of the load-module. The command RETAINSEC is performed after all the object modules for the link operation have been read. The commands RETAINSEC/DELETESEC (see the command DELETESEC) are performed sequentially in the order they are given.
The command RETAINSEC must be used with the EPL options (see OPTIONS EPL) otherwise it has no effect. The command syntax is:



where:

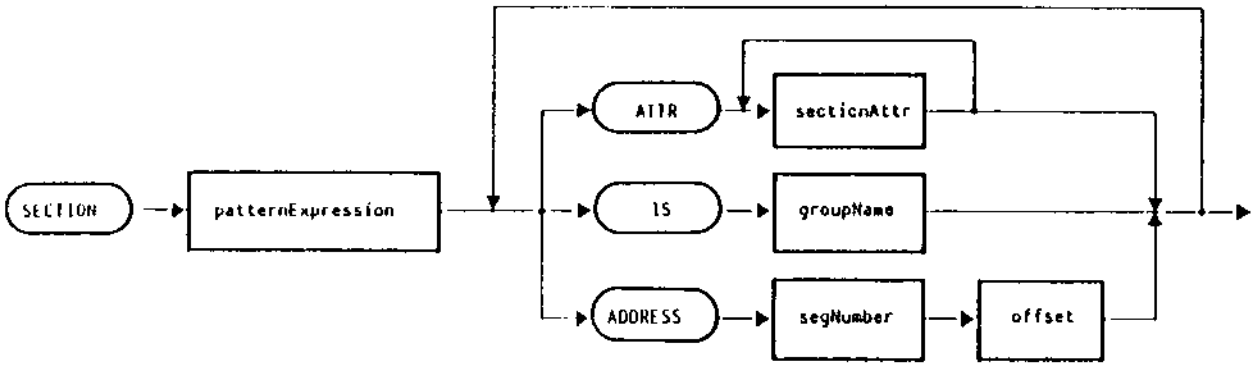
- "patternExpression" is defined as above.

ROOT

Ensures that the root region becomes the current region. This region has one overlay which is loaded into virtual memory each time the program is executed. The command has no parameters.

SECTION

Modifies the attributes, the allocation group assignment and/or the virtual memory assignment of all the sections indicated.
These commands are processed after the Linker has read all the object modules. They are analysed in the same order in which they have been written to the Linker command list.
The command syntax is:



where:

- "PatternExpression" is an expression used to identify the section to be modified. The characters it is composed of assume the following meaning:

  * : identifies zero or more characters of any type, that is any nam
  ? : identifies any single character
  [abks]: identifies one of the characters specified in the square brack
  in this case a,b,k or s.

- "sectionAttr" specifies the attributes of the new section. These may be:

  EXECUTE
  READ
  WRITE
  ABSOLUTE
  RELOCATE
  PRIVATE
  SHARE
  FETCH
  OVERLAY
  CONCATENATE
  INTERPRET
  LINKED
  STACK
  DEBUG

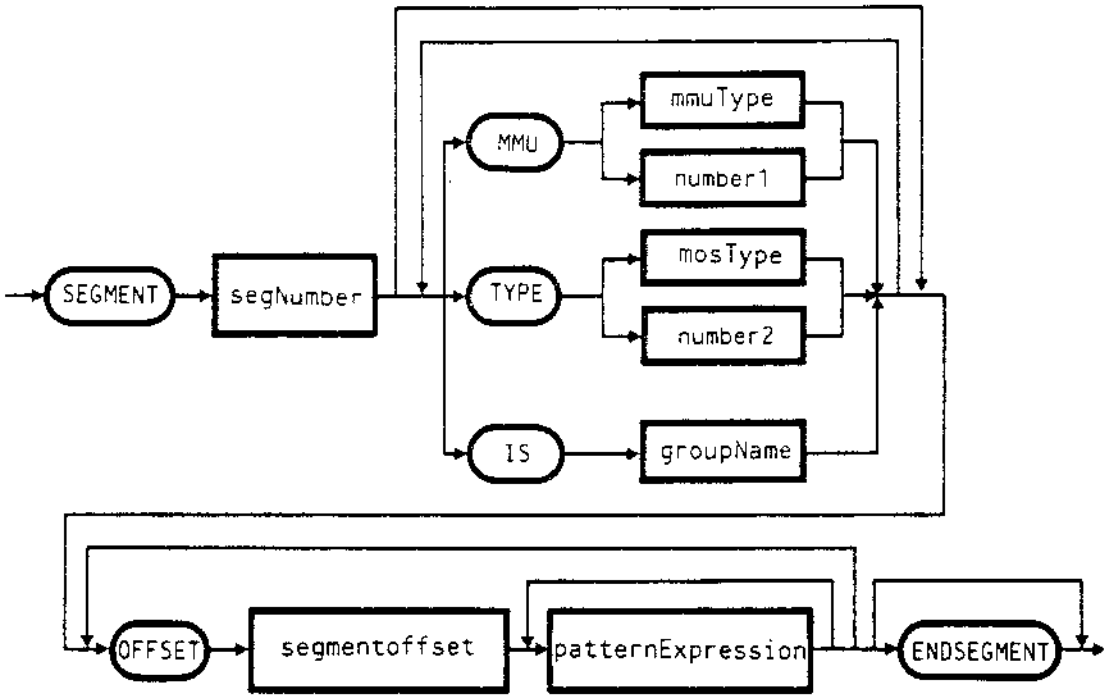
  The attributes not specified are not relevant to this section.

- "groupName" specifies the name of the allocation group to which all the new sections belong.

- "segNumber" and "offset" are the absolute address of the memory for each section.

SEGMENT

Allocates a segment as defined by the user (see the BLOCK-DESCRIPTOR command).
The command syntax is:



where:

- "segNumber" specifies is the segment number and may be in the range 0 to 63

- "number1" is the "mmuType" (RDWRT,RDONLY,XQTONLY,STCKATTR) code (0,1,8,32), which represents one of the following segment attributes:

  0 : RDWRT - segment which may be accessed for read, write or execute operations
  1 : RDONLY - segment which may be accessed for read and execute operations
  8 : XQTONLY - segment which may be accessed only for execution
  10 : STCKATTR- segment having stack function.

- "number2" represents the "mosType" code which may have one of the following values:

```
1 : XQTCODE  - code for execution only
2 : RDCODE   - code for reading only
3 : RWCODE   - code for reading and writing
4 : RDDATA   - data which may only be read
5 : RWDATA   - data on which read and write operations may be executed
6 : STACK    - stack segment
```

When assigning values to "number1" and "number2", the following limitations on compatibility must be respected:

number1	number2
0	3, 5
1	2, 4
8	1
32	6

- "groupName" specifies the name of the allocation group to which the new sections belong.

- "segmentoffset" is a number which specifies the segment offset.

- "patternExpression" is an expression used to identify the section. The characters it is made up of have the following meaning:

```
    * : identifies zero or more characters of any type, that is
        any name
    ? : identifies any single character
[abks]: identifies one of the characters specified in square
        brackets in this case a,b,k or s.
```

SEPARATE

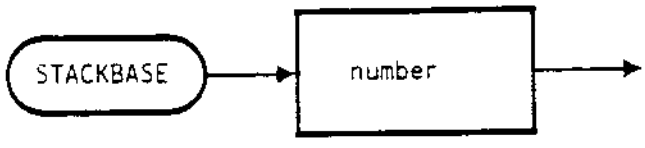
Indicates that the overlay structure, defined in the object module, must be built using a different virtual memory segment for each overlay.
The command has no parameters.

SQUEEZE

This command is compatible with the ZLOC Linker but has no effect on the OLINK Linker.

STACKBASE

This enables the user to modify the base of the stack. Olink assigns a default hexadecimal value to the base of the stack, which is FFFE. This command is essential to the link phase with languages that have a stack-base other than the one assigned by default by Olink.
The command syntax is:



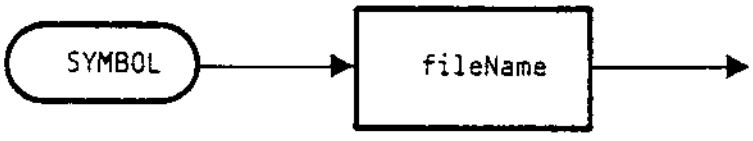
STATISTICS

Causes the program to produce statistical information on the amount of Linker memory that has been occupied.
This command is compatible with the ZLOC Linker.
It has no parameters and is equivalent to the following option:

```
OPTIONS    XREF    MODULE LIST    MEMORY MAP
```

SYMBOL

Specifies that a symbol file must be created. This is a binary file which contains those symbol names and their absolute physical addresses which may be useful for successive link operations or for the debugger. The created file has the same format as EPL (see the command OPTIONS). The syntax is:

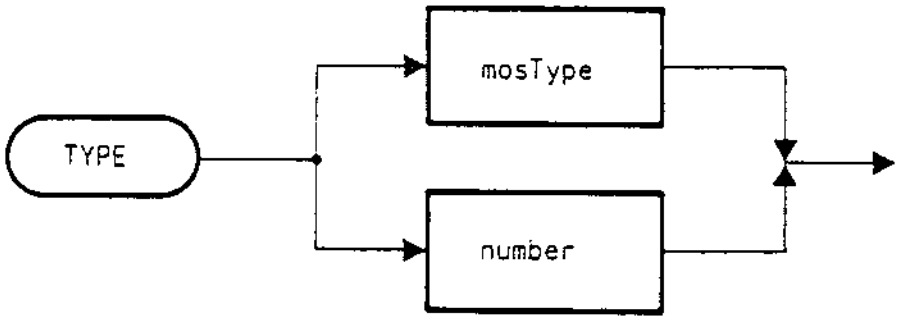


where:

'filename" identifies a symbol file.
The file is identified by its pathname, which follows the same conventions as the Shell environment.

TYPE

Assigns a type to the segment created and informs the system of its contents and specifies the operations which may be carried out on it.
This command may recur any number of times.
It's syntax is:



where:

- "number" represents the segment type code, which may be used as an alternative to "mosType". It may assume one of the following values:

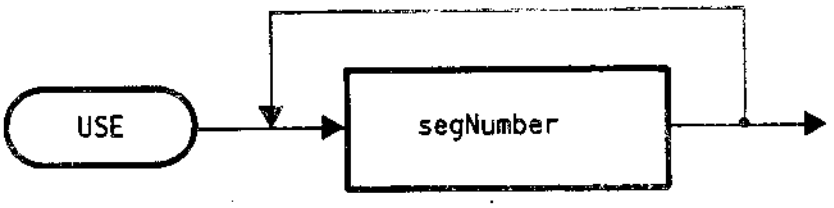
```
1 : XQTCODE   - code for execution only
2 : RDCODE    - code for read operations only
3 : RWCODE    - code for read/write operations
4 : RDDATA    - data which may be read only
5 : RWDATA    - data on which read/write operations may be
                executed
6 : STACK     - stack segment
```

USE

This command specifies a list of segment numbers which may be used by the Linker for the current region. The following must be specified in the default file so that the Linker does not use the system segments:

```
REGION SYSTEM1 USE 0 1 2 3 ... 37 38 39 62
```

and those segments are reserved for the system. The USE command, if present, is always associated with the REGION command (see the REGION command). The two together do not, however, imply that the specified region is used for allocating segments, but simply limits the number of segments that can be used by other defined regions. The command syntax is:



where:

- "segNumber" specifies the segment number and may assume a value in the range 0 to 127.

EXAMPLES

This section gives some examples for using the command language, in order to show the features of the general Linker.

For examples of the dynamic link see COBOL - Program Preparation and Execution

Example 1: Including Modules

The following command is used to chain a program, starting from a single object file:

```
OLINK filename
```

the following command is used to chain a program in a file, and to search a library:

```
OLINK filename libname
```

If the Entry Point List file is wanted in the resulting load module, with all the root's symbols, the following command is used:

```
OLINK filename libname OPTIONS EPL RETAIN "*"
```

(The * character is in inverted commas for SHELL reasons.)

If the Internal SYmbols Dictionary is to be suppresses, and the file is to be loaded in a special file, identified by its pathname, the following command is used:

```
OLINK filename libname OPTIONS EPL RETAIN "*" NO ISD PROGRAM pathname
```

Example 2: Defining the Allocation Groups

The following command is used to allocate all sections which do not have a specific suffix (e.g. "-tbl"):

```
GROUP table NAME *_tbl
```

If no attribute was defined in the definition, sections will be inserted in the group on the basis of their name only. If a separate segment for all data sections having the "share" attribute is required, the following definition is to be provided.

```
GROUP sharedata SIZE 64
                ATTR SHARE NO EXECUTE NO STACK WRITE
                TYPE RWDATA MMU RDWRT
```

Example 3: Section Management

If the forced allocation of all section having the "_shr" suffix into the allocation group shown in the previous example is required, use the following command:

```
SECTION *_shr IS sharedata
```

If, in addition, all shared sections are required to have an overlay structure and be stored in segment 43, the following command is used:

```
SECTION *_shr IS sharedata ADDRESS 43 0 ATTR OVERLAY
```

Example 4: Building Libraries

The Linker program for building files is called by providing a file list to be included in the new library:

```
BUILD libraryname file1 file2 file3 ...
```

Example 5: Building Overlays and Defining Output Required

This Linker command file is an example of an overlay structure and defines the following output: the memory map, the cross reference list and the module list.

```
# the resulting load module will be contained in the file LKLK.A
  PROGRAM LKLK.A

# output requested: memory map, cross-reference list e module list
  OPTIONS MEMORY XREF NO SILENT MODULE

# current region:"root" containing the two specified modules
  ROOT

# main
          LKLKA.O
# subprogram
          LKLKB.O

# current region:"alpha" made up of two overlays
  REGION alpha

# the "alpha" region must reside in segments 44 and 45
  USE     44 45

# definition of the first overlay made up of two modules:
  OVERLAY beta

# module called from LKLKA.O
          LKLKX.O
# module called from LKLKX.O
          LKLKY.O

# definition of the second overlay made up of one module called by
# LKLKA.O
  OVERLAY gamma
          LKLKZ.O

# current region:"root". All the sections whose names start with
# FI_*, US_*, ST_* are assigned to it
  ROOT
          ASSIGN FI_*
          ASSIGN US_*
          ASSIGN ST_*
```

Example 6: Linker Output

An example of "cross-reference list", "module list" and "memory map" follows.

The **cross-reference list** specifies the name, the address, the module which defined the symbol and those which use it for each symbol.
There are two further sections at the end of "cross-reference list":

- one for undefined symbols, in which the name, the module that references it, the pathname of the file that references it, are specified

- one for statistics, in which the number of defined symbols and the number of references is shown.

```
- OLINK 7.1 -      - MOS REL. 5.1 -      CROSS REFERENCE LISTING

SYMBOL NAME     VALUE          DEFINED BY    REFERENCES
    :              :
c_getparam      (<31>)382E     c_init
c_ignore        (<31>)8C3C     visc
c_init_exlb     (<30>)1982     c_init        c_callan
c_init_exub     (<30>)1980     c_init        c_callan
c_init_linksectnum
                (<30>)077E     c_init        c_callan
c_init_linksectptr
                (<30>)077A     c_init        c_callan
c_init_ovinf    (<30>)07AC     c_init
c_starte        (<31>)8A98     stio
c_stdopened     (<31>)818e     nucl          c_stop
c_stopprog      (<31>)4430     c_init        c_stop
c_stopsort      (<31>)89EA     nucl          c_callan  c_stop    c_init
c_string        (<31>)81E6     nucl
c_suppress      (<31>)88D6     rowr
c_terminate     (<31>)88E2     rowr
c_terminit_bottomline
                (<31>)81F2     nucl          c_stop
c_terminit_c_setwsinfo

    :


- OLINK 7.1 -      - MOS REL. 5.1 -      UNDEFINED SYMBOLS

SYMBOL              REFERENCING    REFERENCING FILE
 NAME                 MODULE          PATH NAME



- OLINK 7.1 -      - MOS REL. 5.1 -      STATISTICS

DEFINED             REFERENCES
SYMBOLS             TO SYMBOLS

 593                    241
```

The **module list** specifies the pathname of the file which contains each module, each module name and its version, the name of the section in which each module has been loaded, its size and the name of the group to which the respective attributes belong.
At the bottom of the module list there is a part of statistics which contains the number of sections that have been defined.

```
- OLINK 7.1 -       - MOS REL. 5.1 -       MODULE LIST

  FILE      MODULE        SECTION          SIZE     GROUP        ATTRIBUTES
PATH NAME   NAME          NAME                      NAME         COMMENTS

:B44:232 O
            cobol232 -- cobol VERSION 507 85 10:24 10 +9 35
                        SK_:B44:232000 0000     STACK        WRITE READ PRIVAT RELOC
                                                             CONCAT STACK
                        ME_:B44:232001 0002     DATA         WRITE READ PRIVAT RELOC
                                                             OVRLAY
                        FI_:B44:232002 00E0     DATA         WRITE READ PRIVAT RELOC
                                                             CONCAT
                        US_:B44:232003 0500     DATA         WRITE READ PRIVAT RELOC
                                                             CONCAT
                        ST_:B44:232004 0150     DATA         WRITE READ PRIVAT RELOC
                                                             CONCAT
        :                  :

 IPL DPO :OB_RTE:LIB/SORT.UI.OBJ
            sort -- VERSION 00
                        sort_0          00A0    CODE_CONST   PRIVAT RELOC CONCAT
                        sort_1          0000    DATA         PRIVAT RELOC CONCAT
 IPL DPO:OB_RTE:LIB/RPWR.UI.OBJ
            rpwr -- VERSION 00
                        rpwr_0          0042    CODE_CONST   PRIVAT RELOC CONCAT
                        rpwr_1          0000    DATA         PRIVAT RELOC CONCAT
 IPL DPO :OB_RTE LIB/VISA.UI.OBJ
            visa -- VERSION 00
                        visa_0          02EE    CODE_CONST   PRIVAT RELOC CONCAT
                        visa_1          0000    DATA         PRIVAT RELOC CONCAT




- OLINK 7.1 -       - MOS REL. 5.1 -       STATISTICS

DEFINED SECTIONS

 104
```

The **memory map** provides information about the state of the memory area. The following information is provided for each section which is contained in a segment and possibly in an overlay: memory start and end address, flag for the initialisation of the memory area, section name and respective module name. The type of each segment is also specified.
At the end of the memory map there is a summary of the segments used in the current link.

```
- OLINK 7.1 -       - MOS REL. 5.1 -       MEMORY MAP

(REGION)   SEGMENT MEMORY AREA FLAG SECTION        MODULE
 OVERLAY           START  END  INIT  NAME           NAME

(ROOT)

           (30)                                                DATA TYPE  38
                  0000   0001       ME_0844232001 cb44232
                  0002   00ED YES   FI_0844232002 cb44232
                  00EE   05F9 YES   US_0844232003 cb44232
                    :                   :             :
                  216e                lmsgtbl_a     lmsgtbl
                  216e                                       (CONCATENATED
                                      aux_a         aux      216e
                                      aux_a         aux      216e
                  216e                cre_a         cre
                  216e                c_rtintid_a   c_rtintid
                  2164                nucl_a        nucl
                  216e                rtio_a        rtio
                  216e                sqrn_a        sqrn
                  2164                sort_a        sort
                  2164                rpwr_a        rpwr
                  216e                visc_a        visc

           (31)                                                CODE_CONST TYPE  34
                  0000   00a1 YES   CN_0844232005 cb44232
                  0062   008F YES   CD_0844232006 cb44232
                    :                   :             :
                  88DA   88AB YES   sort_p        sort
                  88AC   88ED YES   rpwr_p        rpwr
                  88EE   8ED8 YES   visc_p        visc

            3F                                                STACK TYPE  3e
                  FFFE               SK_0844232000 cb44232
                  FFFE               c_init_s      c_init
                  FFFE               c_stop_s      c_stop
                    :                   :             :
(SYSTEM1)
           (00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
           17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 3E

 (SYSTEM2)
           (40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56
           57 58 59 5A 5B 5C 5D 5E 5F)

(NOMMU2)
           (60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76
           77 78 79 7A 7B 7C 7D 7E 7F)

(USERPACKAGE)
           (28 29 2A 2B 2C 2D 2E 2F)

- OLINK 7.1 -       - MOS REL. 5.1 -       SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES

  30 31)          (ROOT                 (30, 216e (31  8ED0 (3F  FFFE
```

Example 7: Dynamic Link

In the example that follows there are external data that are exported to the next link step, together with the directive RETAINSEC and DELETESEC. The EPL is generated for the ROOT link unit and is given in input to the links that follow.

All the link units used different segments (the USE directive defines the segments that cannot be used in the current link). This example also contains part of the module list and of the memory map.

The module list contains all the section that have been defined, including the shared ones.

the shared sections are not shown in the memory map, because they are the in EPL file, and consequently are not allocated in the segments of the current program.

This example is valid for Release 5.1: for Release 5.2, the files RTLINK.ROOTLU, RTLINK.INTLU, RTLINK.OVLU must be used instead of the file RTLINK.
For further details see COBOL - Program Preparation and Execution

```
OLINK PROGRAM CB44131.E OPTIONS NO SILENT EPL MEMORY RETAIN  *
      RETAINSEC  *
      DELETESEC  _cob*
      CB44131.O /IPL/DPC/COB_RTS/LIB/RTLINK

OLINK PROGRAM CB44131C1.E OPTIONS NO SILENT MEMORY MODULE
      CB44131C1.O CB44131.E/EPL

OLINK PROGRAM CB44131C2.E OPTIONS NO SILENT MEMORY
      REGION A USE 40 41 42
      CB44131C2.O CB44131.E/EPL

OLINK PROGRAM CB44131C3.E OPTIONS NO SILENT MEMORY
      REGION A USE 40 41 42 43 44 45
      CB44131C3.O CB44131.E/EPL
```

```
- OLINK 7.1 -      - MOS REL. 5.1 -      MODULE LIST


  FILE    MODULE      SECTION        SIZE    GROUP        ATTRIBUTES/
PATH NAME  NAME        NAME                   NAME         COMMENTS

CB44131C1.O
          cb44131c1 -- cobol VERSION 609 85/10/31 17:22:56
                       SK_CB44131C10000000   STACK        WRITE READ PRIVAT RELOC
                                                          CONCAT STACK
                       ME_CB44131C10010002   DATA         WRITE READ PRIVAT RELOC
                                                          OVRLAY
                       print-file    00BC                 PRIVAT ABS OVRLAY LINKED
                       US_CB44131C10030086   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       comm-glo      0002                 PRIVAT ABS OVRLAY LINKED
                       comm3-glo     0002                 PRIVAT ABS OVRLAY LINKED
                       comm4-glo     0002                 PRIVAT ABS OVRLAY LINKED
                       error-counter 0004                 PRIVAT ABS OVRLAY LINKED
                       tabglo        003C                 PRIVAT ABS OVRLAY LINKED
                       IN_CB44131C10090002   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       FI_CB44131C10100030   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       CN_CB44131C10110026   CONSTANT     READ SHARE RELOC CONCAT
                       ST_CB44131C10120002   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       CD_CB44131C10130046   CODE         FETCH EXEC SHARE RELOC
                                                          CONCAT
                       CD_CB44131C1014052A   CODE         FETCH EXEC SHARE RELOC
                                                          CONCAT
                       _cobol_ovlnum 0004    CONSTANT     READ SHARE RELOC CONCAT
                       _cob_dummysec 0002    CONSTANT     READ SHARE RELOC OVRLAY
                       _cobol_progtab0026    CONSTANT     READ SHARE RELOC CONCAT
                       _cobol_ptbllen0022    CONSTANT     READ SHARE RELOC OVRLAY
                       _cob_srnname  0062    CODE         FETCH EXEC SHARE RELOC
                                                          OVRLAY
                       _cob_srhblks  0020    CONSTANT     READ SHARE RELOC OVRLAY
                       _cob_srhtmp   0022    DATA         WRITE READ PRIVAT RELOC
                                                          OVRLAY
                       CD_CB44131C10220008   CODE         FETCH EXEC SHARE RELOC
                                                          CONCAT
                       CB44131C1023  00B1    CONSTANT     READ SHARE RELOC CONCAT
                       CB44131C1024  000C    CONSTANT     READ SHARE RELOC CONCAT
CB44131.E/EPL
          CB44131.E -- OLINK 7.1   VERSION 00
                       CB44          0000                 PRIVAT ABS OVRLAY LINKED
                       CB44          0000                 PRIVAT ABS OVRLAY LINKED
                       CB44          0000                 PRIVAT ABS OVRLAY LINKED
                       ME_CB44131001 0000                 PRIVAT ABS OVRLAY LINKED
                       print-file    0000                 PRIVAT ABS OVRLAY LINKED
                       comm-glo      0000                 PRIVAT ABS OVRLAY LINKED
                       comm3-glo     0000                 PRIVAT ABS OVRLAY LINKED
                       comm4-glo     0000                 PRIVAT ABS OVRLAY LINKED
                       test-results  0000                 PRIVAT ABS OVRLAY LINKED
                       error-counter 0000                 PRIVAT ABS OVRLAY LINKED
                       tabglo        0000                 PRIVAT ABS OVRLAY LINKED



- OLINK 7.1 -      - MOS REL. 5.1 -      STATISTICS

DEFINED SECTIONS

 36
```

```
- OLINK 7.1 -       - MOS REL. 5.1 -       MEMORY MAP

<REGION>/ SEGMENT MEMORY AREA FLAG SECTION        MODULE
 OVERLAY          START  END  INIT  NAME           NAME

<ROOT>

          <<28>>                                                DATA TYPE: 05
                   0000   0001       ME_CB44131C1001
                                                   cb44131c1
                   0002   0087 YES   US_CB44131C1003
                                                   cb44131c1
                   0088   0089       IN_CB44131C1009
                                                   cb44131c1
                   008A   00B9 YES   FI_CB44131C1010
                                                   cb44131c1
                   00BA   017B YES   ST_CB44131C1012
                                                   cb44131c1
                   017C   019D       _cob_srhtmp   cb44131c1

          <<29>>                                                CONSTANT TYPE: 04
                   0000   0025 YES   CN_CB44131C1011
                                                   cb44131c1
                   0026   0029 YES   _cobol_ovlnum
                                                   cb44131c1
                   002A   002B YES   _cob_dummysec
                                                   cb44131c1
                   002C   0051 YES   _cobol_progtab
                                                   cb44131c1
                   0052   0073 YES   _cobol_ptbllen
                                                   cb44131c1
                   0074   0093 YES   _cob_srhblks  cb44131c1
                   0094   0144 YES   CB44131C1023  cb44131c1
                   0146   0151 YES   CB44131C1024  cb44131c1

          <<2A>>                                                CODE TYPE: 02
                   0000   0045 YES   CD_CB44131C1013
                                                   cb44131c1
                   0046   056F YES   CD_CB44131C1014
                                                   cb44131c1
                   0570   05D1 YES   _cob_srhname  cb44131c1
                   05D2   05D9 YES   CD_CB44131C1022
                                                   cb44131c1

          <<3F>>                                                STACK TYPE: 06
                   FFFE              SK_CB44131C1000
                                                   cb44131c1

<SYSTEM1>
          <<00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
          17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 3E>>

<SYSTEM2>
          <<40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56
          57 58 59 5A 5B 5C 5D 5E 5F>>

<NOMMU2>
          <<60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76
          77 78 79 7A 7B 7C 7D 7E 7F>>
```

```
- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
(<30 31>>         ROOT               (<30>>213C (<31>>770A (<3F>>FFFE



- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
(<28 29>>         ROOT               (<28>>019E (<29>>0714 (<3F>>FFFE



- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
(<2B 2C>>         ROOT               (<2B>>019E (<2C>>0706 (<3F>>FFFE



- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
(<2E 2F>>         ROOT               (<2E>>0150 (<2F>>0624 (<3F>>FFFE
```

In the example that follows there are external data and files that are exported to the next link step, together with the directives RETAINSEC and DELETESEC. The EPL is generated for the ROOT link unit and is given in input to the links that follow.
All the link units used different segments (the USE directive defines the segments that cannot be used in the current link). This example also contains part of the module list and of the memory map.
The module list contains all the sections that have been defined, including the shared ones.

The shared sections are not shown in the memory map, because they are in the EPL file, and consequently are not allocated in the segments of the current program.

```
$LINK PROGRAM CB44131.E OPTIONS NO SILENT EPL MEMORY RETAIN  \
      RETAINSEC  *
      DELETESEC  _copy*
      CB44131.O  IPL/DPO/OOB_RTS'LIB/PTLINK

$LINK PROGRAM CB44131C1.E OPTIONS NO SILENT MEMORY MODULE
      CB44131C1.O CB44131.E/EPL

$LINK PROGRAM CB44131C2.E OPTIONS NO SILENT MEMORY
      REGION A USE 40 41 42
      CB44131C2.O CB44131.E/EPL

$LINK PROGRAM CB44131C3.E OPTIONS NO SILENT MEMORY
      REGION A USE 40 41 42 43 44 45
      CB44131C3.O CB44131.E'EPL
```

```
- OLINK 7.1 -      - MOS REL. 5.1 -      MODULE LIST


  FILE     MODULE       SECTION        SIZE    GROUP        ATTRIBUTES
PATH NAME   NAME         NAME                   NAME         COMMENTS

CB44131C1.O
          cb44131c1 -- cobol VERSION 609 85/10/31 17:22:56
                       SK_CB44131C10000000   STACK        WRITE READ PRIVAT RELOC
                                                          CONCAT STACK
                       ME_CB44131C10010002   DATA         WRITE READ PRIVAT RELOC
                                                          OVRLAY
                       print-file    0080                 PRIVAT ABS OVRLAY LINKED
                       US_CB44131C10030086   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       como-glo      0002                 PRIVAT ABS OVRLAY LINKED
                       como3-glo     0002                 PRIVAT ABS OVRLAY LINKED
                       como4-glo     0002                 PRIVAT ABS OVRLAY LINKED
                       error-counter 0004                 PRIVAT ABS OVRLAY LINKED
                       toglo         003C                 PRIVAT ABS OVRLAY LINKED
                       IN_CB44131C10090002   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       FI_CB44131C1010003C   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       CN_CB44131C1011002e   CONSTANT     READ SHARE RELOC CONCAT
                       ST_CB44131C10120002   DATA         WRITE READ PRIVAT RELOC
                                                          CONCAT
                       CD_CB44131C10130046   CODE         FETCH EXEC SHARE RELOC
                                                          CONCAT
                       CD_CB44131C1014002A   CODE         FETCH EXEC SHARE RELOC
                                                          CONCAT
                       _cobol_ovlnum 0004    CONSTANT     READ SHARE RELOC CONCAT
                       _cob_dummysec 0002    CONSTANT     READ SHARE RELOC OVRLAY
                       _cobol_progtab0026    CONSTANT     READ SHARE RELOC CONCAT
                       _cobol_otblen0022     CONSTANT     READ SHARE RELOC OVRLAY
                       _cob_srnname  0062    CODE         FETCH EXEC SHARE RELOC
                                                          OVRLAY
                       _cob_srhdlr   0020    CONSTANT     READ SHARE RELOC OVRLAY
                       _cob_srntmo   0022    DATA         WRITE READ PRIVAT RELOC
                                                          OVRLAY
                       CD_CB44131C10220008   CODE         FETCH EXEC SHARE RELOC
                                                          CONCAT
                       CB44131C1023  0081    CONSTANT     READ SHARE RELOC CONCAT
                       CB44131C1024  0000    CONSTANT     READ SHARE RELOC CONCAT
CB44131.E/EPL
          CB44131.E -- OLINK 7.1   VERSION 00
                       CB44          0000                 PRIVAT ABS OVRLAY LINKED
                       CB44          0000                 PRIVAT ABS OVRLAY LINKED
                       CB44          0000                 PRIVAT ABS OVRLAY LINKED
                       ME_CB44131001 0000                 PRIVAT ABS OVRLAY LINKED
                       print-file    0000                 PRIVAT ABS OVRLAY LINKED
                       como-glo      0000                 PRIVAT ABS OVRLAY LINKED
                       como3-glo     0000                 PRIVAT ABS OVRLAY LINKED
                       como4-glo     0000                 PRIVAT ABS OVRLAY LINKED
                       test-results  0000                 PRIVAT ABS OVRLAY LINKED
                       error-counter 0000                 PRIVAT ABS OVRLAY LINKED
                       toglo         0000                 PRIVAT ABS OVRLAY LINKED


- OLINK 7.1 -      - MOS REL. 5.1 -      STATISTICS

DEFINED SECTIONS

 96
```

```
- [illegible] -        - MOS REL 3.1 -        MEMORY MAP

REGION/  SEGMENT MEMORY AREA FLAG SECTION          MODULE
OVERLAY          START  END  INIT  NAME             NAME

ROOT

         <2B>                                                    DATA TYPE  03
                 0000   0001      ME_CB4413101001
                                                     cb44131c1
                 0002   0087 YES  US_CB4413101003
                                                     cb44131c1
                 0088   0089      IN_CB4413101009
                                                     cb44131c1
                 008A   00B9 YES  FI_CB4413101010
                                                     cb44131c1
                 00BA   0178 YES  ET_CB4413101012
                                                     cb44131c1
                 0170   0190      _cob_snntmp    cb44131c1

         <29>                                                    CONSTANT TYPE  04
                 0000   0025 YES  CN_CB4413101011
                                                     cb44131c1
                 0026   0029 YES  _cobol_ovlnum
                                                     cb44131c1
                 002A   002B YES  _cob_dummyseed
                                                     cb44131c1
                 002C   0051 YES  _cobol_progtab
                                                     cb44131c1
                 0052   0073 YES  _cobol_otbllen
                                                     cb44131c1
                 0074   0093 YES  _cob_snnblks   cb44131c1
                 0094   0144 YES  CB4413101023   cb44131c1
                 014C   0151 YES  CB4413101024   cb44131c1

         <2A>                                                    CODE TYPE  02
                 0000   0045 YES  CD_CB4413101013
                                                     cb44131c1
                 0046   056F YES  CD_CB4413101014
                                                     cb44131c1
                 0570   05D1 YES  _cob_snnname   cb44131c1
                 05D2   05D9 YES  CD_CB4413101022
                                                     cb44131c1

         <<3F>>                                                  STACK TYPE  06
                 FFFE             SK_CB4413101000
                                                     cb44131c1

SYSTEM1>
         <<00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
         17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 3E>>

SYSTEM2>
         <<40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56
         57 58 59 5A 5B 5C 5D 5E 5F>>

NOMMU2>
         <<60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76
         77 78 79 7A 7B 7C 7D 7E 7F>>
```

```
- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
(30 31)           ROOT               (30)0130 (31)770A (3F)FFFE



- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
 (28 29)          ROOT               (28)019E (29)071A  (3F)FFFE



- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
 (28 2D)          ROOT               (28)019E (2D)070A (3F) FFFE



- OLINK 7.1 -      - MOS REL. 5.1 -      SEGMENT SUMMARY

SEGMENTS          REGION    OVERLAY  SIZES
(2E 3F)           ROOT               (2E)0150 (3F)0A04 (3F) FFFE
```

4. ERROR MESSAGES

This chapter describes the OLINK error messages. The value of the %STATUS variable can be tested in the SHELL environment to find out whether the link has been correctly executed.

- %STATUS = 0 correct end
- %STATUS = 1 warning
- %STATUS = 2 fatal errors (abort)

The error messages are contained in the map. A signal is given at the point where the error occurs; the recovery action is specified with the warning messages, and abort for the fatal errors.

"xx...x" IS MULTIPLY DEFINED

The symbol "xx...x" has been defined already.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove error, recompile and relink

"xx...x" IS NOT AN OBJECT FILE / LOAD FILE / LIBRARY FILE / INSYM FILE / COMMAND FILE

The specified symbol is not an object file/a load file/ a library file/ an INSYM file/a command file

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove error and relink
- check that the following files are not still open:

VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

"xx...x" MUST BE A LIBRARY FILE

The specified symbol must be a library file

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove error and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

ABSOLUTE SECTION "xx...x" ASSIGNED TO REGION "yy...y" USED SEGMENT <<nn>>
REGION "zz...z" ALREADY USES THAT SEGMENT

Conflict in assignment of segments to the indicated regions. The segment <<nn>> is not available for the region "yy...y".

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove error and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

BAD MMU VALUE IN A SEGMENT DIRECTIVE -- MMU VALUE IGNORED

Incorrect MMU value in a SEGMENT directive. The value of MMU is ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove error and relink

BAD OFFSET VALUE IN A SECTION DIRECTIVE

Incorrect OFFSET value in a SECTION directive.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove error and relink

BAD SEGMENT NUMBER AFTER SEGMENT DIRECTIVE

Segment number incorrect after a SEGMENT directive.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

BAD TYPE VALUE IN A SEGMENT DIRECTIVE -- TYPE VALUE IGNORED

Incorrect TYPE value in a SEGMENT directive. The value is ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error on the command file and relink

CANNOT FIND GROUP "nn"

The specified GROUP cannot be found.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

CANNOT FIND SYMBOL "xx...x" -- ENTRYPOINT DIRECTIVE IGNORED

The specified symbol cannot be found. The ENTRYPOINT directive is ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

CANNOT OPEN FILE "xx...x"

The specified file cannot be opened.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check that the file "xx...x" exists and that it is correct
- check the File System integrity using DISKCHECK and VOLGC utility
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

CANNOT OPEN LOAD MODULE FILE "xx...x"

The specified load-module cannot be opened.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check that the file "xx...x" exists and that it is correct
- check the File System integrity using DISKCHECK and VOLGC utility
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

COMMAND FILE OVERFLOW

Overflow in command file.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: reduce the command file to the maximum size of 32 Kbytes and relink

---

DUPLICATE OBJECT MODULE "xx...x" -- READ IN ANYWAY

Duplicate object module name.
If this error is followed by the message ""'symbol name" IS MULTIPLY DEFINED" for all the symbols defined in the module, it means that the specified module has actually been duplicated.
If the display of the error is not followed by the message ""'symbol name" IS MULTIPLY DEFINED", it means that two different modules exist with the same name.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: none

ERROR FOUND IN THE INSYM FILE "xx...x"

Error in the INSYM "xx...x" file.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action:
- check the correctness of the "xx...x" file
- check File System integrity using DISKCHECK and VOLGC utilities

---

ERROR: REORGING FILE

An error has occurred when reading/writing on an OLINK workfile.
The error may have been caused by:

- an inconsistent File System
- data corrupted by another process
- an Olink error

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check File System integrity using DISKCHECK and VOLGC utilities
- relaunch the linker
- contact the Olivetti Software Maintenance Service
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

ERROR WHILE LOADING/UNLOADING LINKER OVERLAYS SYSTEM COMMAND: "xx...x" RETURNED ERROR: CLASS "yy...y" NUM "nnnn" PRIOR OVERLAY: "OVxx" NEW OVERLAY: "OVyy"

Error during loading/unloading linker overlays.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- take action consistent with:
ERROR: CLASS "yy...y" NUM "nnnn" specified in the message
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FATAL ERROR - LINKER ABORTED

Fatal error: the linker terminates execution.

Recipient: programmer

Class: fatal

Consequences: the linker aborts. Execution is terminated because of the previous errors.

Action:
- remove errors and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FATAL ERROR: SYMBOL TABLE OVERFLOW IN THE "xx...x" CLASS ... TRY TO REDUCE

The symbol table in the "xx...x" is full.

Recipient: programmer

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- reduce program dimensions according to the advice given in the Program Preparation manual for the language which you are using. Use of CHECK is advised.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FORMAT ERROR IN "xx...x": USAGE OF SYMBOL BEFORE SYMBOLREF ENTRY

Incorrect object format. The error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FORMAT ERROR NEAR "nnnn": BAD SECTION -- <<FF>>FFFF USED

Incorrect object format. The error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action: - check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FORMAT ERROR NEAR "nnnn": BAD SYM REF -- <<FF>>FFFF USED

Incorrect object format. The error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action: - check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FORMAT ERROR NEAR "nnnn": EVALUATION STACK OVERFLOW

Incorrect object format. The error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FORMAT ERROR NEAR "nnnn": SHORT ADDRESS > 255 -- REDUCED MODULO 256

Incorrect object format. The error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

FORMAT ERROR NEAR "nnnn": TEXT OUTSIDE SECTION "xx...x" -- IGNORED

Incorrect object format. The error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

INTERNAL LINKER ERROR - ABORT

OLINK internal error that has occurred during load-module generation.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O: ATTEMPT TO REUSE I/O BUFFERS

OLINK internal error: too many I/O buffers.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O: BAD ACCESS RIGHTS ON FILE: "name"

Internal OLINK error: access rights to the file called "name" have been set incorrectly.

Recipient: programmer

Class: recoverable

Consequences: OLINK aborts, return to MCL

Action:
- modify access rights to the files and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O: BAD FILE READ SELECTED "nn"

OLINK internal error: a wrong file number has been selected when reading.
"nn" is the OLINK internal number that identifies the file.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O: BAD FILE WRITE SELECTED "nn"

OLINK internal error: a wrong file number has been selected when writing.
"nn" is the OLINK internal number identifying the file.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O: EOF ON "nn"

OLINK internal error: unexpected EOF when reading a file. "nn" is the OLINK internal number that identifies the file.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O ERROR ON VIRTUAL MEMORY WORK FILE

I/O error on VTR.nnnnnnnn workfile.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O: ERROR OPENING FILE: "name"

Error returned by the system primitive OpenByte. The primitive has returned a code other than CORRECT.

Recipient: programmer

Class: recoverable

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O ERROR OPENING WORKFILE DGN.nnnnnnnn

Error returned during generation of the file DGN.nnnnnnnn

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the File System integrity using DISKCHECK and VOLGC utilities
- reactivate the linker
- check free space on the volume: it must be enough to contain the files VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn where 'nnnnnnnn' is the process number

---

I/O ERROR OPENING WORKFILE VRT.nnnnnnnn

Error returned during generation of the file VRT.nnnnnnnn

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the File System integrity using DISKCHECK and VOLGC utilities
- reactivate the linker
- check free space on the volume: it must be enough to contain the files VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn where 'nnnnnnnn' is the process number

---

I/O ERROR READING OBJECT LIBRARY FILE

I/O error when reading a library object file.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O ERROR WHILE CLEARING/CREATING LOAD MODULE DIRECTORY "xx...x"

I/O error when deleting/creating the load-module "xx...x" directory.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O ERROR WHILE EXTENDING VIRTUAL MEMORY WORK FILE -- NO MORE SPACE?

I/O error while extending the pagination file of the virtual memory for the Symbol Table: VRT.nnnnnnnn .

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O ERROR WHILE READING OBJECT MODULE NEAR "nnnn"

I/O error when reading an object module.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O ERROR WHILE WRITING TO LOAD MODULE FILE

I/O error when writing to a load-module.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O FATAL ERROR READING FILE "nn"

Error returned by the system primitive ReadByte. The primitive has returned a reply code other than CORRECT. "nn" is an internal number of the linker that identifies the file in which the error occurred.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O FATAL ERROR WRITING FILE

Error returned by the system primitive WriteByte. The primitive returns a reply code other than CORRECT.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check File System integrity using the DISKCHECK and VOLGC utility
- reactivate the linker
- contact the Olivetti Software Maintenance Service
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number
- check free space on the volume: it must be enough to contain the files VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn where 'nnnnnnnn' is the process number

---

I/O: OUT OF FILES

OLINK internal error: too many files opened simultaneously for reading and/or writing.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- reduce the nesting level of the command file
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O POSITIONING ERROR ON "nn"

OLINK internal error: incorrect positioning in file.
"nn" is an internal number of the linker that identifies the file in which the error occurred.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O UNABLE TO ESTABLISH LOCAL ROOT DIRECTORY

Error returned by the system primitive CONNECT. The primitive returned a reply-code other than CORRECT.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

I/O UNABLE TO ESTABLISH WORKING DIRECTORY

Error returned by the system primitive CONNECT. The primitive has returned a reply-code other than CORRECT.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

INVALID OBJECT FORMAT

Incorrect object format. This error is caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action: - check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

"xx...x" IS MULTIPLY DEFINED

The symbol "xx...x" has been defined already.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove error, recompile and relink

"xx...x" IS NOT AN OBJECT FILE / LOAD FILE / LIBRARY FILE / INSYM FILE / COMMAND FILE

The specified symbol is not an object file/ a load file/ a library file/ an INSYM file/ a command file

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

LOAD MODULE "xx...x" HAS CONFLICTING SEGMENT WHITH "yy...y"

There is a conflict between the segments of load-module "xx...x" and those of load-module "yy...y".

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

MISSING ADDRESS AFTER ADDRESS PARAMETER OF A SECTION DIRECTIVE

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error in the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

MISSING GROUP NAME AFTER GROUP PARAMETER OF A SECTION DIRECTIVE

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error in the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

MISSING OFFSET VALUE FOR ABSOLUTE ENTRYPOINT -- ENTRY POINT IGNORED

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

MISSING OVERLAY NAME AFTER OVERLAY DIRECTIVE

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error in the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

MISSING PATH NAME AFTER PROGRAM DIRECTIVE -- DIRECTIVE IGNORED

The PROGRAM directive is not followed by a pathname. Directive ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

MISSING REGION NAME AFTER REGION DIRECTIVE

Self-explanatory

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error in the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

MISSING SECTION NAME AFTER SECTION DIRECTIVE -- DIRECTIVE IGNORED

Self-explanatory

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

MISSING SECTION PATTERN AFTER ASSIGN DIRECTIVE

Self-explanatory

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error in the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

MMU VALUE INCOMPATIBLE WITH TYPE VALUE

Self-explanatory

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

"xx...x" MUST BE A LIBRARY FILE

The specified symbol must be a library file

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

NAME SHOULD BE FOLLOWED BY A SECTION NAME PATTERN -- DIRECTIVE IGNORED

The section pattern in the GROUP directive is missing after the key name NAME. Directive ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

NO GROUP FOUND FOR SECTION "xx...x" -- GROUP "yy...y" USED

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

NO GROUP NAME AFTER GROUP DIRECTIVE -- DIRECTIVE IGNORED

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

NO LOAD MODULE, USE "LINK" AS PROGRAM DIRECTORY

This is a message: it is displayed when the PROGRAM directive does not occur.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: none

NO MORE SEGMENTS AVAILABLE

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- try to reduce the size of the link-unit
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

NO NUMBER FOLLOWS SIZE PARAMETER OF A GROUP DIRECTIVE -- IGNORED

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

NO OVERLAYS ALLOWED IN ROOT

The OVERLAY directive is only allowed if a region other than ROOT is active.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error in the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

PROGRAM DIRECTORY ALREADY EXISTS

This message is displayed when the program directory exists already and the NO REPLACE option is set.

Recipient: system manager

Class: recoverable

Consequences: execution continues and a request for a new program directory name followes

Action: supply new name for the program directory

READ ERROR: ID "nn" AT LOC "fileoff"

An error has been found when reading from disk, in the offset "fileoff" of the file "nn" containing the pagination for the Symbol Table. "nn", "mm", "fileoff" are internal OLINK numbers. The error may depend on:

- an inconsistent File System
- data corrupted by another process
- an Olink error

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check File System integrity using the DISKCHECK and VOLGC utilities
- reactivate the linker
- contact the Olivetti Software Maintenance Service
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

READ ERROR: WANTED ID: "nn" GOT "mm" AT LOC "fileoff"

The identifier "nn" has been searched for when reading from disk, but "mm" has been found instead at location "fileoff". ""nn" "mm" and "fileoff" are internal OLINK numbers.

The error may have been caused by:

- an inconsistent File System
- data corrupted by another process
- an Olink error

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check File System integrity using the DISKCHECK and VOLGC utilities
- reactivate the linker
- contact the Olivetti Software Maintenance Service
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

REFERENCE TO MULTIPLY DEFINED SYMBOL "xx...x" -- IGNORED

Reference to a symbol that has been defined already. Ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error in the command file and relink

REFERENCE TO UNRESOLVED SYMBOL "xx...x" -- <<FF>>FFFF USED

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error from the source file, recompile and relink

---

SECTION "xx...x" HAS CONFLICT BETWEEN CONCATENATE/OVERLAY ATTRIBUTE

The attributes CONCATENATE/OVERLAY are in conflict in section "xx...".

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error from the command file and relink

SECTION "xx...x" IS OVERLAID WITHOUT HAVING OVERLAY ATTRIBUTE -- IGNORED

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error from the command file and relink

SECTION "xx...x" IS TOO LARGE TO ALLOCATE

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- reduce the size of the specified section to the permitted maximum of 64 Kbyte. Then recompile and relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

SECTION "xx...x" WAS NOT ASSIGNED TO A REGION/OVERLAY -- ASSIGNED TO ROOT

Self-explanatory.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: if the action undertaken by the linker is not wanted, modify the input commands

---

SEGMENT "nn" MULTIPLY RESERVED -- IGNORED

Two REGIONS have tried to reserve the same segment using the directive USE. The second one is ignored.

Recipient: programmer

Class: recoverable

Consequences: execution continues

Action: remove the error from the command file and relink

SEGMENT DIRECTIVE REDEFINES CONTENTS OF SEGMENT "nn"

The SEGMENT directive redefines the contents of segment "nn".

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error from the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

TOO MANY CHARACTERS IN THE TOKENS: TRY TO REDUCE THE CHARACTERS

The total length of all the names and of the input directive keywords (including the default file) cannot be greater than 32 Kbyte.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action: reduce the number of tokens, or their size, and relink

---

UNABLE TO BUILD "LINK" PROGRAM DIRECTORY

When a program directory is created, the call to a system primitive returns a reply-code other than CORRECT.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check the integrity of the File System using the DISKCHECK and VOLGC utilities. Relink.
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

UNABLE TO OPEN DEFAULT FILE

The default file, under the OLINK program directory, is missing.

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- restore the default file in the OLINK program directory

UNBALANCED '[]' IN PATTERN "xx...x"

The '[]' are not matched in the pattern "xx...x"

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- remove the error from the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

UNEXPECTED EOF ENCOUNTERED IN MODULE "xx...x"

Unexpected end of the logical file of the specified module.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the consistency of the module on disk, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

UNEXPECTED EOF ON COMMAND INPUT

End of file occurs before the logical end of the directives file.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the consistency of the command file and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

UNKNOWN OBJECT ENTRY ENCOUNTERED NEAR "nnnn"

Wrong object format. The error has been caused by incorrect code generation.

Recipient: programmer

Class: recoverable

Consequences: execution continues until abort

Action:
- check the correctness of the source file, recompile and relink
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

WRITE ERROR: EXPECTED "nn" FOUND "mm"

An error has occur when writing to an OLINK workfile. "nn" and "mm" are internal OLINK numbers. The error may have been caused by:

- an inconsistent File System
- data corrupted by another process
- an OLINK error

Recipient: system manager

Class: fatal

Consequences: OLINK aborts, return to MCL

Action:
- check File System integrity using the DISKCHECK and VOLGC utilities
- reactivate the linker
- contact the Olivetti Software Maintenance Service
- check that the following files are not still open:
VRT.nnnnnnnn, DGN.nnnnnnnn, MSG.nnnnnnnn
where 'nnnnnnnn' is the process number

PART II - THE GENERALISED SIMBOLIC DEBUGGER

INTRODUCTION TO PART II

Part two describes the characteristics of the generalised symbolic debugger and the commands for its use.

The debugger command language defines the potential of the debugger and is the tool which acts as a user interface.

5. INTRODUCTION

CHARACTERISTICS

The Generalised Symbolic Debugger aims at a complete control of compiled and linked programs in execution, isolating and removing the cause of errors.

It is interactive in that it allows the user to follow the program execution. It examines the program contents allowing the user to alter them if necessary.

The debugger is called symbolic because it is able to reference program locations and the variables via their symbolic names defined in the source program. The contents of a variable may thus be examined without making reference to its address. In the same way it is possible to insert a break point by specifying the name of a procedure, a label, a module or the source program line number.

It is called generalised in that it currently supports two languages: BASIC and COBOL. It distinguishes the different data conversion types to be executed, the mode in which data is to be correctly displayed and the context rules applicable, for each language. The set of information contained in the Internal Symbol Dictionary (ISD) guarantees the transparency of the debugger with respect to the programming languages.

FUNCTIONS VISIBLE TO THE USER

By means of user commands the debugger may:

- examine or change the contents of the memory area which contains the user program via the operating system.

- obtain information from the Internal Symbol Dictionary created by the compiler and Linker. This information allows the user to specify the same symbol identifiers used in the program in the debugging phase. If there is no Internal Symbol Dictionary the user must refer to a symbol by its absolute address which he may find in the program memory map.

Input/Output and Redirection

The debugger, unless otherwise stated, uses standard I/O files of the system (usually the terminal from which it has been called) to receive commands in input and supply output. Input/output of the debugger may however be redirected to other files via the "in" and "out" commands. In this manner the user may, therefore, write his commands, using the text editor, outside the debugger and execute them within the debugger whenever he wishes to. The standard I/O file must however be used to display error messages.

Only the debugger files can be redirected; those for the application program cannot be redirected.
Any existing file specified during redirection of output is overwritten.
Input is redirected by retrieving it from a file, whilst output is redirected by writing it on another file or displaying it on another video.

If the debugger's output, to be displayed on the screen or redirected, will not fit on the screen, the following message is emitted:

```
>> HIT RETURN TO CONTINUE OR ENTER 'Q' TO STOP <<
```

This allows the user to stop or continue display.

CALLING THE DEBUGGER

The debugger has been implemented as a program which may be called from the Shell environment by specifying the following path name:

```
DEB    UserProgramName
```

where:

- "UserProgramName" identifies the program-directory containing the program to be debugged.

The debugger may be used from different work stations concurrently as long as the programs being controlled are different or the same program has a different name for each work station.

USING THE DEBUGGER AS A CALCULATOR

If no "UserProgramName" is specified, the debugger can be used as a desk-top calculator to carry out calculations, conversions, etc. To use this option the "SHOW" command and, if need be, the "SET" command, must be used as shown in the following example:

```
set &a:=1000
show ,h (&a+800)/%12

64
```

The decimal value 800 is added to the value 1000, and the result is divided by the hexadecimal value 12. The final result is displayed in hexadecimal format.

The "QUIT" command is used to return to Shell.

PASSING THE PARAMETERS

If either positional or keyed parameters are to be passed to a user program, the PRG keyword must precede the program name in the command.

```
DEB  PRG=userProgramName par1
```

where:

- "par1" is the positional parameter passed to the user program.

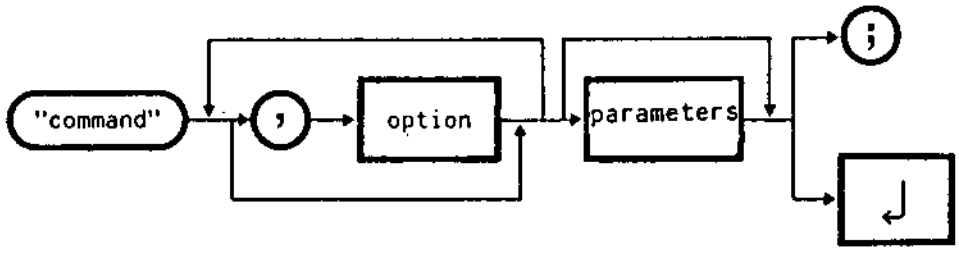
Notes

A program with an overlay structure can be debugged.
Breakpoints can only be set in overlays if the "RUN" command has been given to start program execution.

6. COMMAND LANGUAGE

STRUCTURE

The general command structure is:



where:

- "command" is the command name which may be abbreviated to a minimum of characters that identify the command unequivocally. The first two characters are always sufficient.
- "option" is an optional keyword and usually specifies the mode in which the command will operate.
- "parameters" usually constitutes at least one expression.

Notes

- All commands with the exception of "BEGIN...END" must terminate with ";" or correspond with an EOL.
- Extra spaces which may have been entered between command elements or expressions are ignored.
- A blank must always be put between the name of the command and its parameters and one parameter and another, so as to avoid incorrect interpretation of the command. If the command "AD" is written instead of "AT" the debugger interprets it as an AT command (identified by the letter A) whose argument (D) indicates where the breakpoint is. The lack of a delimiter between the command and its argument causes an error message to be emitted, as opposed to an invalid command error.
- Commands, options, parameters and variables may be entered in upper or lower case letters.

- Expressions are semantic structures which define the computation rules for variables, modules, procedures, program lines or labels and for generating new variable values making use of operators. The contents of a variable are used as its value; the value of a procedure, line number or label is the address.

- The rules which govern the construction of an expression (including the use of round brackets are the same as those used in the main programming languages and define the priority of the various operators.

- Alphanumeric (string) operands are compared one character at a time starting from the leftmost character. Comparison ends when two characters differ or there are no more characters to be compared.

- The boolean "true" and "false" values are represented by the numeric values 1 and 0.

- When the debugger is asked to display a COBOL structured variable and cannot interpret its format, it displays the hexadecimal format by default (typically for displaying the outermost levels).

OPERATORS

The debugger has three types of operators:

- logical operators (NOT, AND, OR)

- arithmetic operators (*,/,+,-)

- relational operators (>,<,=,<=,<>,>=)

Unary operators (+, -) and the NOT logical operator have the highest priority. These are followed by the multiplicative operators (*,/,AND) and additive operators (+,-,OR). Relational operators have the lowest priority. Operators having the same priority are executed starting the right hand side.

The '-' (unary or binary) arithmetic operator must be separated from the operands to which it refers by at least one space on each side.

OPERANDS

Each expression consists of one or more operands connected by arithmetic, logical or relational operators. The debugger checks that:

- arithmetic operators do not apply to alphanumeric operands

- an alphanumeric operand is not compared with any other operand which is not alphanumeric.

The following are the fundamental operands recognised by the debugger.

- numeric constants

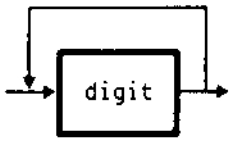
- numeric constants

  . decimal
  . hexadecimal
  . real

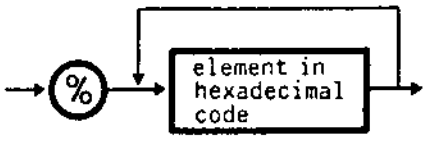
- alphanumeric or string constants
- debug variables
- program variables
- modules
- procedures or functions
- labels
- number of source program lines
- addresses.

The syntax of each operand is given below, an explanation of certain components has been given when necessary.

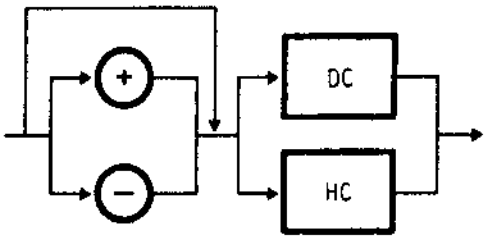
Decimal Constants (DC)



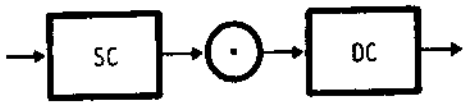
Hexadecimal Constants (HC)



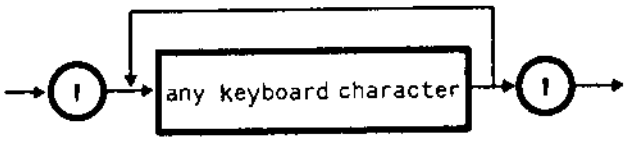
Signed Constants (SC)



Real Constants

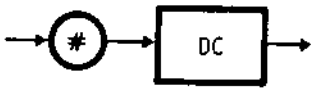


Alphanumeric Constants (AC)

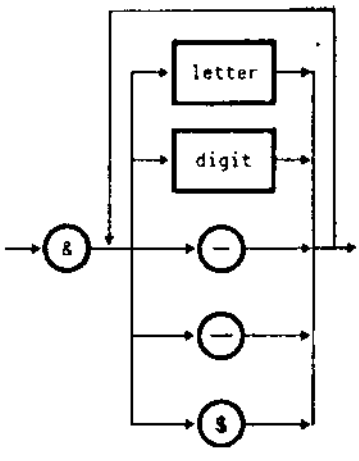


The alphanumeric string within the quotes must be maximum 120 characters long. If quotes are used these must be double quotes (").

Line Number Constants



Debug Variables



Debug variables may be predefined or user defined. In both cases the variable's symbolic name must be preceded by the "&" character.

The predefined variables make it possible to reference:

- The simple registers (&r0, &r1 .. &r15)
- The double registers (&rr0, &rr2 .. &rr14)
- The program counter (&pc)

- The stack pointer (&sp)

- The current segment (&seg).

The user defined debug variables may be real or alphanumeric and may be defined solely via the SET or EQUATE commands (for further information see the "COMMANDS" section). They may be referenced only after they have been defined.

Only the first six characters of the variable's symbolic name are significant.

Program Variables, Module, Procedure/Function and Labels

See the respective language manuals for the syntax of all the program variables, modules, procedures/functions and labels, which are dependent on the program language and which may be referenced in a debugging session.

COMMENTS

Comments may be inserted within debug commands. They must be enclosed by braces (e.g.: {comment}) and are ignored by the debugger.
They are particularly useful for the redirection of debugger input/output as they are contained in the I/O file.

MATRICES

The debugger allows direct access to each component of a matrix (which may be called either vector, table or array) via its symbolic name followed by one or more subscripts separated by commas and enclosed in square brackets. Rows, columns, submatrices or the matrix itself may be referenced by specifying the respective subscript range.

The matrix name followed by a pair of square brackets will cause all the matrix elements to be referenced: MAT[].

To reference the element of a row or column, specify the name of the matrix and the subscripts of the row or column followed or preceded by a comma.
Example: if for a matrix having three rows and four columns the following is specified: MAT[2,] MAT[,3], the elements which will be referenced are: MAT[2,1] ... MAT[2,4] MAT[1,3] ... MAT[3,3].

To reference the elements of a submatrix specify the name of the matrix and the subscript range.
E.g.: if MAT[1:2,3:4] is specified the elements identified are: MAT[1,3] MAT[1,4] MAT[2,3] MAT[2,4].

Nested Indices

In a BASIC program, the debugger limits the nested indices to four nesting levels.

SYMBOLIC NAMES

Each symbolic name defines a single source program element. The hierarchy of program elements varies according to the programming language used. The name of the program (the most external module or context) which contains the element always occupies the highest hierarchy level.

To define a source program element "." must be placed between the identifier and the element being qualified.
The complete symbolic name is only necessary if there is any possibility of ambiguity.

Example

If the INVOICE program contains the following data structure declarations

```
01 CUSTOMER.
   02 NAME.
      03 CODE1 PIC 9(11).
      03 ADDRESS PIC X(30).


01 CODE.
   02 CODE1 PIC 9(11).
   02 CODE2 PIC X(16).
      .
      .
      .
```

the CODE1 field must be referred to as "NAME.CODE1" if it belongs to the CUSTOMER record, and "CODE.CODE1" if it belongs to the CODE record. The ADDRESS field, however, may be referred to simply as ADDRESS.

FUNCTIONS

Each operand has an associated value. If the operand is a constant its value is the constant itself.
The value associated with a variable is its contents whereas that associated with a module, procedure, label and line number is its corresponding address.

Two debugger functions are available for associating the address and program line number to a variable:

- a function which calculates an address
- a function which calculates line numbers.

Function which Calculates an Address

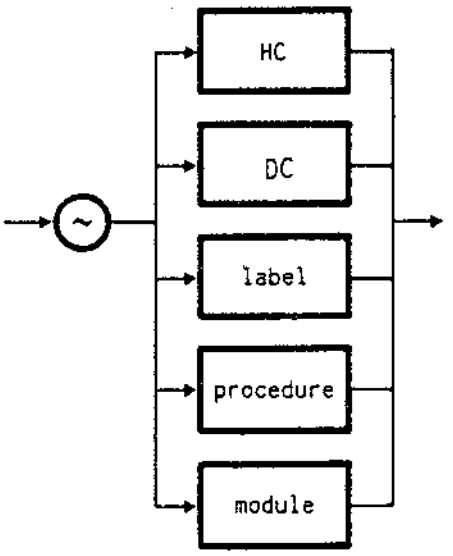
This function transforms the symbolic name of a program variable and returns the corresponding address. The variable's symbolic name must be preceded by the special "@" character.

The function has the following syntax:

The SHOW command must be used to display the address (e.g. show @abc).

Function which Calculates Line Numbers

This function transforms an actual or symbolic address (label, procedure or module) and returns the line number of the source program of that address. The address must be preceded by the special "~" character. The function has the following syntax:



The SHOW command must be used to display the address of a certain number of lines, for example:

```
show #26
```

The address on display always refers to the first executable statement, starting from the specified line. If the program lines 26, 27 and 28 in the example above contain declarations and not executable statements, line 29's address will be displayed.

ADDRESSES

The address formats known to the debugger are as follows:

- procedure name (address of the first instruction in the procedure)
- label (label address)
- line number (address of the first instruction in the line)
- "address function" output (address of the required variable)

- user or local variable that has a numeric value

- explicit address specified as <segment-number>offset.

All the numeric values are interpreted as decimals by default. The hexadecimal values must be preceded by the character "%".

The debugger interprets the numeric values as follows:

%12	0000 0012	<current-segment>%0012
%144	0000 0144	<current-segment>%0144
%2649	0000 2649	<current-segment>%2649
% 45678	0004 5678	ERROR
%115678	0011 5678	ERROR
%2115678	0200 5678	ERROR
%2005678	0200 5678	<%2>%5678
%26335723	2633 5723	ERROR
%24002345	2400 2345	<%24>%2345
18	0000 0012	<current-segment>%0012
291	0000 0123	<current-segment>%1234
4660	0000 1234	<current-segment>%1234
637538868	2600 1234	<%26>%1234

All the numeric values are significant as addresses whose hexadecimal representation is as follows:

XX00 YYYY

where:

XX is a segment number
00 is a mandatory value
YYYY is the offset in the segment

LOGICAL GROUPINGS OF COMMANDS

The following are the logical groupings into which the debugger commands may be divided:

Commands for breakpoint management:

- AT

- CLEAR

Commands which display memory areas, registers, variables, etc.

- DUMP

- LIST

- SHOW

- TRACE

Commands which control the flow of debugging commands:

- BEGIN ... END
- IF ... THEN ... ELSE ...
- QUIT

Commands for the control of the execution and the modification of programs

- EQUATE
- HALT
- RUN
- SET
- STEP

Commands which identify the debugger I/O files:

- INPUT
- OUTPUT

Command which sets or displays the user defined scope:

- SCOPE

Command which assigns the debug variables:

- EQUATE
- SET

Command which displays the command description:

- HELP

Command for exiting from a BEGIN ... END or an IF ... THEN ... ELSE session:

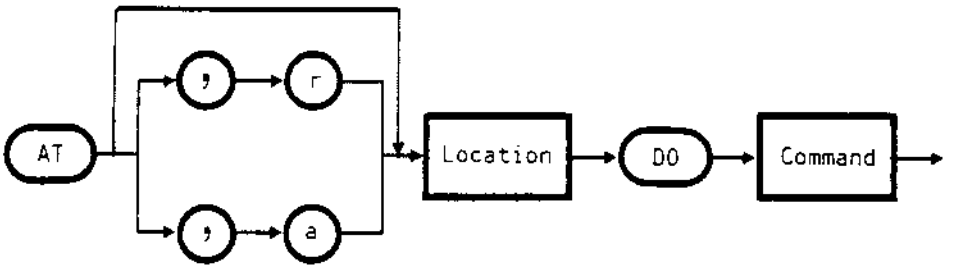
- /CONTROL/ /D/

COMMANDS

The following is a description of all the debugger commands. They are presented in alphabetical order giving the meaning, use and syntax of each command. Examples are given where necessary.

AT

Sets a breakpoint at the specified program point. When the breakpoint is reached the debugger executes the list of debugging commands specified The command syntax is·



where·

- ",a" or ",r" must be specified if the "Location" already contains a breakpoint. If the user specifies:

  ",r" : the old command string is replaced by the new one

  ",a" : the new commands are added at the end of the list (default option).

- "Location" indicates the point at which a breakpoint is to be set. This may be expressed in any of the address formats; it is sometimes represented by an arithmetic expression.

- "Command" identifies the command string to be executed. This is checked immediately for syntax errors.

Note

Following execution of the "AT" command the following are displayed:

- A decimal integer (implant number) which identifies the breakpoint. To remove the breakpoint the user may just specify this number (rather then its address).

- The line number which contains the breakpoint.

- The hexadecimal address of the location.

- The name of the module containing the breakpoint.

- The label (if any) of the specified address (this is displayed both when the breakpoint is set and when it is reached).

If there is no Internal Symbol Dictionary, the line number and the module name containing the breakpoint are not displayed.

Examples

```
at PROCED_1 do if PRICE > 100 then show NET
```

Each time PROCED_1 is encountered the debugger tests the specified condition. If the value of "PRICE" is greater than 100, then the contents of "NET" are displayed.

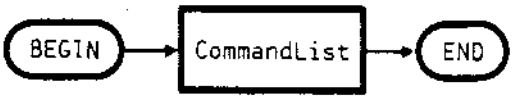
```
at &pc + 14 do ...

at #10 - 4 do ...
```

The values that are added or subtracted refer exclusively to the address offsets. Multiplication and division are not allowed.

See the example in the HALT command.

BEGIN...END

Executes the list of commands contained between BEGIN and END. The debugger does not start executing the command string until it reaches an END. This means that the commands can occupy several input lines. The BEGIN....END command can be used inside the AT, IF...THEN...ELSE and STEP commands. See the /CONTROL/ /D/ command for exiting from a BEGIN..END sequence.
The command syntax is:



where:

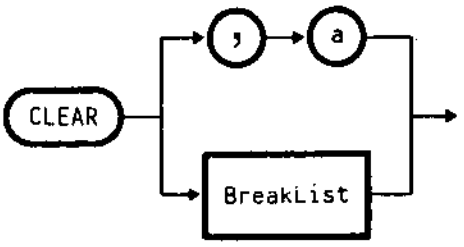
- "CommandList" is the list of debugger commands which may also include another "BEGIN ... END" command.

Note

This command may also be used within the "AT", "IF ... THEN ... ELSE" and "STEP" commands.

CLEAR

Removes one or more breakpoints which have been set. Since the maximum number of breakpoints permitted in any one program is 19, this command may be used to remove old breakpoints so that new breakpoints may be set. The command syntax is:



where:

- "a" is the option which removes all breakpoints currently existing.
- "Breaklist" is a list of positive integers (implant numbers), line numbers or addresses which refer to a breakpoint. These may be in the form of arithmetic expression. Only those breakpoints specified in this list will be removed. The positive integer associated with a breakpoint may be obtained from the message output by the "LIST" or "AT" commands.

Example

```
clear #120,#200
```

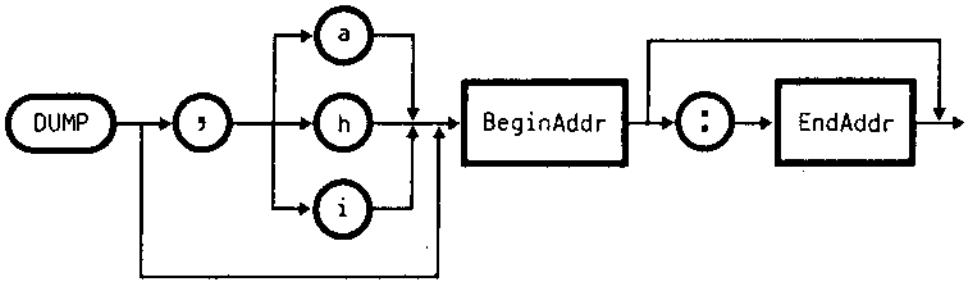
This command removes the breakpoints previously set at lines 120 and 200 of the source program.

```
clear 2,4
```

This command requests the removal of the breakpoints identified by the implant numbers 2 and 4.

DUMP

Displays all or part of the contents of a memory segment.
The command syntax is:



where:

- ",a" displays the memory contents in ASCII characters.
- ",h" displays the memory contents in hexadecimal (default value).
- ",i" displays the memory contents in Assembler Z8000 format.
- "BeginAddr" is the start address of the memory area to be displayed. The address may be entered in the form of an arithmetical expression. The memory address must always be an even number if instructions are to be displayed (option",a" or ",i").
- "EndAddr" is the end address of the memory area to be displayed. The address may be an arithmetical expression. If omitted only 16 bytes will be displayed by default.

Example

```
dump LAB_10:LAB_50
```

The part of the user program starting from the LAB_10 label to the LAB_50 label is displayed in hexadecimal format.

```
dump,i <%40>%100:<>%120
```

Offset 100 ti 120 if segment 40 is displayed in Z8000 Assembler format.

```
dump %150:%390
```

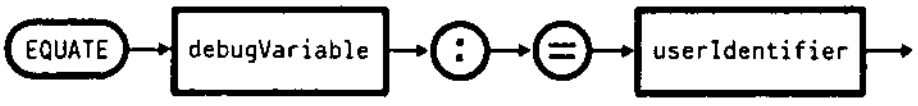
In the above example no segment has been specified, the segment of the user program containing the entry point is therefore assumed by default. The segment contents between offset 150 and 390 (both numbers inclusive and in hexadecimal) are displayed.

```
dump #10:#20
```

This displays the memory area containing the user program lines from 10 to 20 in hexadecimal

EQUATE

Defines a debug variable as an alias for a user identifier.
After an EQUATE command, the only values which may be assigned to the debug variable are those which are consistent with the value of the program variable. If, for example, the program variable is an integer, the debug variable may only take integer values.
The command is particularly useful in cases where one needs to make constant reference to a variable which has a long name.
The command syntax is:



where:

- "debugVariable" is the new name to be assigned to the variable in the debugging phase.
- "UserIdentifier" is the name of the user-defined variable.

Example

```
equate &1:=PROC_1.GROUP_VAR.GROUP_ELEMENT
```

The debug variable '&1' becomes the alias of the GROUP_ELEMENT variable which belongs to GROUP_VAR within the PROC1 procedure.

HALT

Interrupts program execution and requests further debugging commands. If this command is encountered in a command list a message containing the address of the breakpoint and the corresponding program line number is displayed. A "run" command must be issued in order to resume execution. The command has no parameters.

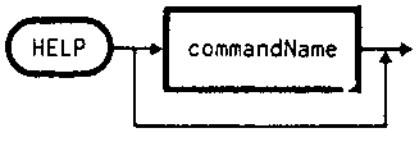
Example

```
at #10 do begin
            set &a:=&a+1
            if &a > 10 then halt
          end
```

The program is interrupted when the &a debug variable's value is greater than 10.

HELP

Displays a brief description of the existing debugger commands.
The command syntax is:

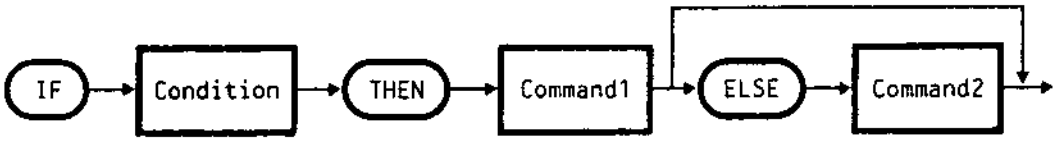


where:

- "commandName" is the command which the user has selected to be described. If omitted the description of all the commands will be displayed in alphabetical order.

IF...THEN...ELSE

Forces the evaluation of a condition. The next action of the debugger depends on the truth value of the specified condition. This may be a single action or a set of actions specified between the BEGIN and END keywords.
The command syntax is:



where:

- "Condition" is a logical expression which returns a boolean value. The expression may contain any type of operator.
- "Command1" indicates the command to be executed if the specified condition is true. This in its turn may be another IF...THEN...ELSE command.
- "Command2" indicates the command to be executed if the specified condition is false, this may be another IF...THEN...ELSE command. It should be noted that if [ELSE....] has been used ELSE must be entered on the same line as Command1.

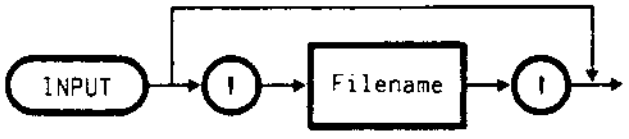
Example

```
at MAIN_PROC DO BEGIN IF X + Y > Z then show X,Y else show Z end
```

Each time the MAIN_PROC procedure is reached the values of X and Y are tested. If the condition is true the variables X and Y are displayed if it is false Z is displayed.

INPUT

Directs the debugger to fetch commands from the specified file. It, therefore, changes the default input (screen).
INPUT commands may be nested in up to 10 levels. When the end of the file is reached, the previous input file is read automatically. When the first input file comes to an end, control returns to the debugger, unless a QUIT command has been specified in the input file.
If the command is given without parameters, the default standard input is reactivated.
The command syntax is:

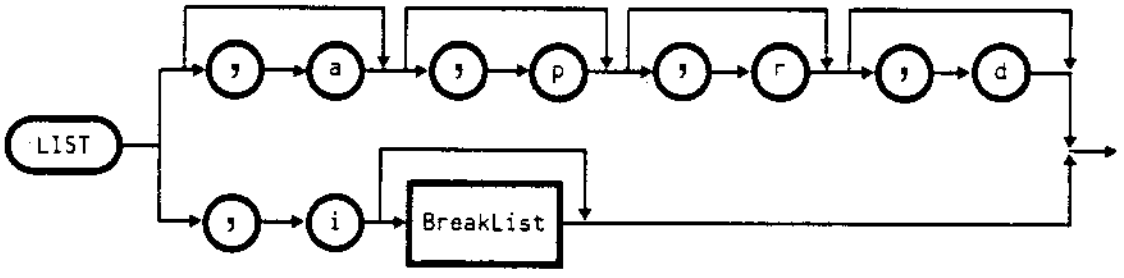


where

- "Filename" is a system file which may be any bytestream file. It is assumed to be under the root directory if '/' is specified at the start, otherwise it is assumed to be under the working directory.

LIST

Displays the status of the current program and the corresponding line number. It also indicates whether the user is operating in "trace" or "step" mode.
The command syntax is:



where at least one of the following options must be specified:

- ",a" displays all the information possible.

  ",p" displays the list of the procedures currently active.

- ",i" displays information about existing breakpoints. If "BreakList" is also specified only those breakpoints specified in the list will be displayed.

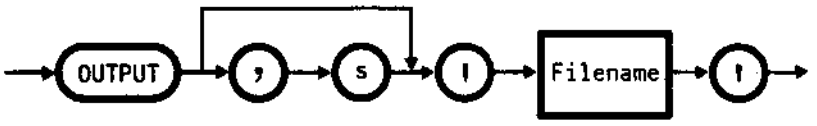
  ",d" displays the list of user defined debug variables.

  "r" dumps user process registers.

  "BreakList" is a list of expressions which define a set of breakpoints.

OUTPUT

The OUTPUT command is used to change the standard output file (screen). It indicates where the subsequent commands are to be written and where their output is to be displayed
The command syntax is:



where:

- ",s" suppresses output on the screen, and directs it to the specified file. If this option is not specified the output will be written on both the screen and the file.

- "Filename" is the name of the system file which may be any bytestream file or the screen to which the output is to be written. If the name starts with "/" the current directory is assumed to be the root directory otherwise the working directory is assumed to be the current directory.

Note

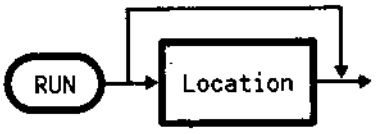
- If there is no existing output file, the debugger will create a byte-stream file.
  If no parameters are specified output is redirected to the screen the last output file specified by the user is deleted.

QUIT

Ends the current debugging session after killing the user process.
The command has no parameters.

RUN

The RUN command reactivates the execution of a user program, starting from the current address of any other address specified by the user. br
The command syntax is:



where:

- "Location" is an arithmetic expression, the evaluation of which provides an address. It indicates the point at which the process is to resume. If this is not specified, the process will be resumed at the point in which it was last interrupted.

Example

```
at #20 do begin if errcode <> 0 then run #100 end
```

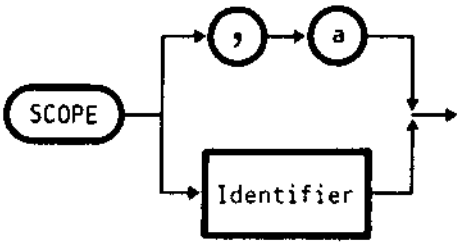
The RUN command in this example enables the user to bypass the execution of source line numbers 20 to 100 if the specified condition is true.

The above example does not work in the current release. To overcome this limitation the user must use the following commands:

```
at #20 do begin halt if errcode <> 0 then run #100 end
```

SCOPE

Sets or displays the user-defined scope used to resolve internal program symbols. When used with no parameters the command displays the current user defined scope.
The command syntax is:



where:

- ",a" sets the maximum scope possible, that is, all program modules will be searched during program resolution. This is the situation which prevails at the start of a debugger session.
- "Identifier" identifies the particular module or procedure to be debugged.

Examples

```
scope SUBR_1.PROC_2
```

The above command changes the scope of symbol resolution of PROC_2 in module SUBR_1. To display the VAR_1 variable, the user may enter the following command:

```
show VAR_1
```

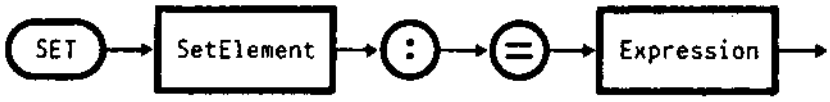
this is the equivalent of:

```
show SUBR_1.PROC_2.VAR_1
```

which must be used if VAR_1 is defined more than once in SUBR_1, otherwise the debugger displays the error message *DEB051* AMBIGUOUS SYMBOL.

SET

Changes the contents of the debugging variable or the variables of the program whose execution is being checked, assigning the value of the right element to the left element.
The command syntax is:



where:

- "SetElement" is a variable, a register or an address.

- "Expression" is an arithmetic expression or a character string. When an arithmetic expression is assigned to "SetElement" the system verifies its type and if necessary it will convert it to that of "SetElement"

Notes

- If "SetElement" is an address (absolute address, program line number, label, module or procedure) then its type is assumed to be a one word binary integer. It must thus be converted to binary before it is assigned to "SetElement".
  If the length of the "Expression" operands exceeds one word, only the rightmost word is stored at the specified address. the rest of the bytes are discarded. If it is only one byte, only the higher address of "SetElement" is changed, the lower address is filled with zeros.
  If "Expression" is a character string, "SetElement" is filled starting from the left. Any remaining space is filled in with blanks.

- If "SetElement" is a register, "Expression" is converted to binary and stored in the register. If "Expression" is numeric the register is loaded from the right, zeros are added to the left. If it is a character string, the register is loaded starting from the left and blanks are added If the right hand side if necessary.

- If "SetElement" is a variable it may have a numeric or character data type. If "Expression" is numeric its value is converted, if necessary, to the same data type as "SetElement" if this is numeric before loading it. If "Expression" is a character string and "SetElement" is numeric or vice versa , an error message is sent out and the contents of "SetElement" remain unchanged.
  If both "SetElement" and "Expression" are character strings and "Expression" exceeds the number of characters in "SetElement", the extra characters to the right are truncated. If "Expression" has less characters than "SetElement" blanks are inserted to the right.

Examples

```
set <25>%10:=%8d07
```

In the above example the hexadecimal constant "8d07" is loaded into offset 10 of segment 25.

```
set &rr2 := 'ABCD'
```

In this example the string "ABCD" is loaded into the double register rr2.

```
set &a:=1
```

The &a variable is given a value of 1, so that it can be used as a cycle counter for example.

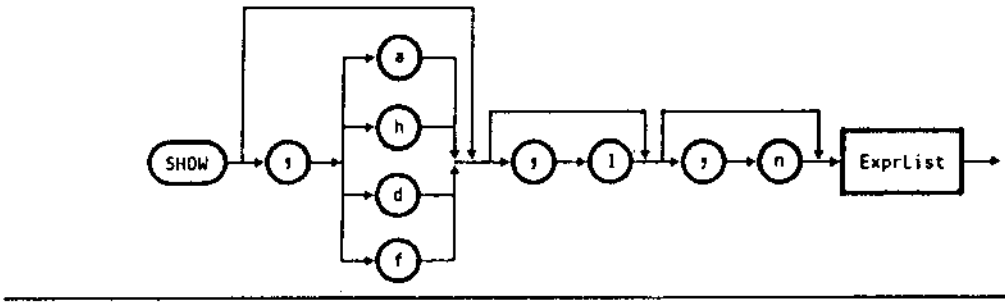
```
set VAR_PROG:=10
```

The VAR_PROG variable is given a value of 10.

SHOW

Displays the contents of a symbol defined in the symbol file (ISD), the contents of the registers, the value of constants, addresses of internal program elements, program line numbers or labels. If no option is specified the element is displayed with its type and structure defined in the source language; under COBOL, a multi-level data structure is displayed in ASCII code.

More than one element may be displayed on one screen line.
The command syntax is:



where:

- ",a" displays the contents in ASCII characters.
- ",h" displays the contents in hexadecimal values.
- ",d" displays the contents in fixed point decimal code.
- ",f" displays the contents in floating point decimal code.
- ",n" displays the name of the element in front of its value.
- ",l" displays each element on a separate line.
- "ExprList" is a list of arithmetic expressions or character strings. Each expression or string is separated by a comma.

STEP

Executes a specified number of source program lines before the user process is suspended. Its effect is the same as when a series of debugger commands is inserted at the start of each source program line.
If no command is specified the debugger stops at every line, and executes a set of commands (insertion of breakpoints) that are not transparent to the user. If this happens debugging is slowed down considerably.
The command syntax is:

where:

- ",e" can only be used for debugging BASIC application programs which include exception handling.
- "StepCount" specifies the number of source program lines which are to be executed before the program is suspended.
- "Command" specifies a debugger command sequence. These are to be executed at the beginning of each source program line.

Example

```
step 10 do begin show X,Y

if A>10 then show REC_POINT end
```

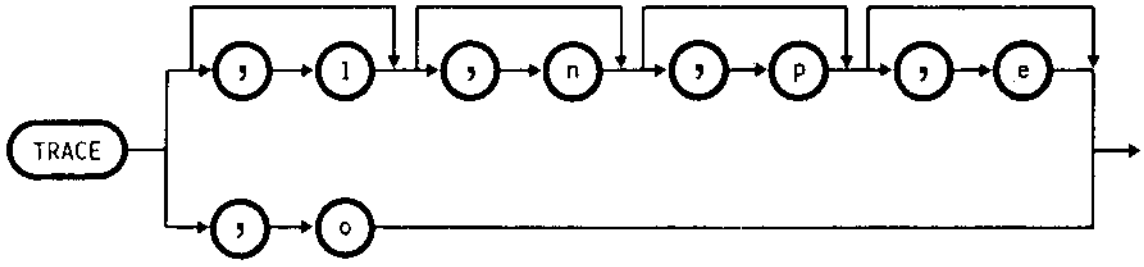
The command in the above example causes the execution of line 10 of the source program. X and Y are displayed before each line is executed. If the condition is true the contents of the "REC_POINT" variable are displayed.

```
at #20 do halt
```

Supposing that the current line is 10, this command executes 10 lines without carrying out any operations.

TRACE

Traces the program execution. If no option is specified all the source program line numbers, labels and procedure names encountered when the program is in execution are displayed.
The command syntax is:



where:

- ",l" displays the program labels.
- ",n" displays the source program line numbers.
- ",p" displays the names of the program procedures.
- ",e" can only be used for BASIC application programs which include exception handling.
- ",o" disactivates an existing TRACE command.

Examples

```
trace,1; run
```

This command will display all the program labels encountered during program execution.

```
at #10 trace
at #100 trace,o
run
```

The above command will trace the program execution from line 10 to line 100, displaying the source program line numbers, and the labels and names of the procedures. The command is disactivated at the start of line 100.

/CONTROL/ /D/

This command is used to exit from a BEGIN or IF session without waiting for it to terminate.
The command must be given at the beginning of the line, otherwise it will not be recognised.
The commands contained in the session are removed starting from the first BEGIN or IF command that has not been closed, so the /CONTROL/ /D/ command must be used very carefully.

Examples

```
at #10 do begin
sh varA
sh varB
if varC=1
/CONTROL/ /D/
```

The structure is annulled as from and including the "at" command.

```
begin
sh varA
if varB>varA then begin
sh varB+4
/CONTROL/ /D/
```

The structure is annulled as from the first "begin".

7. INFORMATIVE OR ERROR MESSAGES

All the messages produced by the debugger are written to standard output files. When redirection of output has been requested (OUTPUT command with option ",s"), the output produced is written to a specified file, or displayed on another screen.
The messages can be informative or relate to semantic or syntactic errors.

The messages displayed by the debugger that are not self-explanatory are listed in alphabetic order below. A description of the error or an explanation of what has happened is given for each message, and the user is given brief advice.

The error messages with prefixes other than "*DEB" (not described in this chapter) refer to internal debugger errors. The Olivetti Software Maintenance Service must therefore be called.

CURRENT ADDRESS: <%xx>%xx...x

Displays the current address of the program and specifies the segment number and the offset.

CURRENT ADDRESS IS IN A MODULE WITHOUT ISD FILE

The current address is of a module for which there is no ISD.

CURRENT PROCEDURE: aa...a

Displays the name of the current procedure.

DEBUGGER IS RUNNING WITHOUT USER PROGRAM

No user program name has been specified: the debugger may only be used as a desk-top calculator.

FIRST AVAILABLE SEGMENT: <%xx>

The number identifying the first segment available in memory for the user is displayed in hexadecimals.

INSTALLATION ERROR - BAD OR MISSING DEBUGGER MESSAGE FILE

Error message emitted in initialisation. The DEB/err.d file has been installed incorrectly or is missing.

INSTALLATION ERROR - BAD OR MISSING ISD HANDLER

Error emitted in initialisation.
The DEB/bt/MAIN file has been installed incorrectly or is missing.

LIST OF ALLOCATED LOCAL VARIABLES: VAR TYPE EQUATE TO

aa...a	bb...b	cc...c
.....	.....	.....

A list of allocated local debug variables is listed. The following information is given for each variable: its name, its type and if the EQUATE command has been specified, the name of tne element that the variable has been equated with.

PROGRAM LOAD POINT ADDRESS: <%xx>%xxxx

The address of the program loading point has been supplied; this consists in a segment number and an offset.

STOPPED AT ADDRESS: IMPLANT nn...n aa...a nn...n <%xx>%xxxx

The implant number for the breakpoint at which the program has stopped has been supplied with the name of the program, the number of the line at which it has stopped (if the ISD is present) and the address, specified as a segment number and an offset.

USAGE: DEB PRG= <PROGRAM-DIRECTORY> [PARAMETERS]

The parameter have been supplied without specifying the key word "PRG=".

WALKBACK SEQUENCE:

The list of all the active procedures is supplied.

DEB001 WRITE MEMORY ERROR

Writing in memory not allowed.
Error returned following a system call (PMM).

DEB002 READ MEMORY ERROR

Reading in memory not allowed.
Error returned following a system call (PMM).

DEB003 INPUT-FILE STACK OVERFLOW

More than 10 input files have been specified.

DEB006 ISD FILE READ ERROR - CLASS: XX CODE: YY

The ISD is incorrect: it must be reconstructed.
See the "Message Book", Appendix A for the explanation of system error code "XXYY".

DEB007 USER REGISTERS READ ERROR

The registers cannot be read.
For example, the user program has finished and the contents of the registers is inconsistent.
Error returned following a system call (PMM).

DEB008 USER REGISTERS WRITE ERROR

Reading in register not allowed.
For example the reading request is incompatible with the register format.
Error returned following a system call (PMM).

DEB009 SET IMPLANT ERROR

Incorrect attempt at inserting a breakpoint.
If the message is emitted in initialisation,
followed by:
DEB069 ERROR LOADING USER PROGRAM-CLASS:01 CODE:16
this means that the PMM has been configured without
the debugger modules: it must be reconfigured.

DEB010 INPUT_FILE OPEN ERROR

The input file cannot be opened.
For example the specified file does not exist.

DEB012 OUTPUT FILE OPEN ERROR

The output file cannot be opened.
Error emitted following an attempt at redirection
of output.

DEB013 OUTPUT-FILE CREATE ERROR

The output file cannot be created.
Error emitted after an attempt at redirecting output.

DEB014 INPUT FILE READ ERROR

The input file cannot be read.
Error emitted after an attempt at redirecting input.

DEB015 OUTPUT FILE WRITE ERROR

Writing not allowed on the output file.
Error emitted following an attempt at redirecting
output onto a file where this is not allowed.

DEB016 WARNING: REDUNDANT OPTIONS

Incompatible options have been specified.

DEB021 LINE NOT FOUND

The program line has not been found.

DEB022 ILLEGAL EXPRESSION

Specified expression incorrect.

DEB027 CAN'T ADD TWO ADDRESSES

Two addresses cannot be added together because they are specified as "segment-offset", and only offsets can be added together or an offset added to an address.

DEB028 CAN'T SUBTRACT TWO ADDRESSES WITH DIFFERENT SEGMENTS

Two addresses can only be subtracted within the same segment.

DEB029 CAN'T SUBTRACT ADDRESS FROM NUMBER

An address cannot be subtracted from a number because the address is specified as "segment-offset" and only offsets can be subtracted.

DEB030 CAN'T MULTIPLY ADDRESS

An address cannot be multiplied because it is made up of "segment-offset" and only offsets can be multiplied.

DEB031 CAN'T DIVIDE ADDRESSES

An address cannot be divided because it is made up of "segment-offset" and only offset can be divided.

DEB032 ZERO DIVISOR

Division by zero illegal.

DEB034 NOT USABLE ISD FILE

The ISD file is incorrect; it must be reconstructed.

DEB036 STRING TOO LONG

The string exceeds 120 characters.

DEB037 INVALID COMMAND

Non-existent command.

DEB039 STRING NOT CLOSED

The alphanumeric string has not been closed with a single quote.

DEB040 MISSING DELIMITER

The delimiter ";" or /CR/ for commands is missing or a blank character is missing between the command and its parameters and between the parameters themselves.

DEB044 ERROR OCCURRED WHEN EXECUTING THE USER PROCESS

The user program has terminated.
An error has occurred at run-time.

DEB047 MUST BE ALPHABET OR NUMERIC DIGIT

Special characters cannot be used in this context (e.g. identifier).

DEB048 COMMENT NOT CLOSED

Comment has not been closed: delimiter missing.

DEB049 MUST BE HEXADECIMAL DIGIT

Digits must be hexadecimal (0-F).

DEB050 MUST BE NUMERIC DIGIT

Characters must be numeric (0-9).

DEB051 AMBIGUOUS SYMBOL

The symbol (line number, label, variable etc.) has been defined for more than one module: the SCOPE must be used to differentiate it, or the symbol specified using the module that contains it:
e.g. "module.line-number".
For structured variables the ambiguous symbol must be referred together with the element that qualifies it unequivocally (e.g. "bills.customer").

DEB052 ILLEGAL CHARACTER OR CHARACTER IN WRONG PLACE

Identifier written incorrectly.

DEB056 ONLY PROCEDURES ALLOWED

Wrong argument in the SCOPE command: only modules and procedures are allowed.

DEB058 CONVERT REAL TO LONG OVERFLOW

Overflow in conversion from a real number to a 4 byte integer.

DEB061 SYSTEM ERROR - CLASS:XX CODE:YY

One of the following "XXYY" class system code errors has been emitted:

- If the program that is running contains an overlay, an error can have occurred in the overlay table loaded by the run-time language.

- Too many processes are active or too many channels are being used: check how many processes and how many channels have been defined in the configuration file. This error message may be displayed during initialisation and causes the debugging to terminate.

- Terminal accessing error: check terminal state or, if possible, activate the debugger from another terminal; this can be emitted both during initialisation and when the debugger is active.

If the message is transmitted when the user program has finished, check if the corresponding process is still active; if it is, it must be killed from the Shell environment.

For the meanings of error code "XXYY" see the "Message Book", Appendix A.

DEB062 CANNOT RUN USER PROGRAM

The same program has been activated from two terminals. The error message is displayed on the second one.

DEB065 ERROR IN OVERLAY HANDLING

Error displayed when there is a request for use of an overlay.
Unfruitful attempts have been made to set two breakpoints in the debugger for handling overlays in the user program. Overlays are not handled.

DEB069 ERROR LOADING USER PROGRAM - CLASS:XX CODE:YY

System initialisation error, regarding either the user program or the terminal.
See the "Message Book", Appendix A, for an explanation of error code "XXYY".

DEB071 FLOATING OVERFLOW/UNDERFLOW

An overflow or an underflow has occurred during a floating point operation.

DEB073 CONVERT DOUBLE TO SINGLE PRECISION OVERFLOW

Overflow during conversion from double to single precision.

DEB081 OVERLAY TABLE NOT YET INITIALIZED

No breakpoint can be inserted at the moment.
A RUN command must have been executed previously.

DEB082 INTERNAL ERROR: BAD RULES FILE DETECTED IN TOK.ACTION

Internal debugger error (TOK.ACTION).
Call Olivetti Software Maintenance Service.

DEB083 LOADING USER OVERLAY - CLASS:XX CODE:YY

System error whilst loading an overlay.
See the "Message Book", Appendix A, for an explanation of error "XXYY".

DEB084 UNLOADING USER OVERLAY - CLASS:XX CODE:YY

System error while unloading an overlay.
See the "Message Book", Appendix A, for an explanation of error "XXYY".

DEB088 ERROR GENERATING ISD TABLE

An error has occurred in generation of the ISD table.
The operation must be repeated.
The source ISD must be saved before activating the debugger so as not to have to recompile and link it if there is a system crash or a power failure.

DEB089 PROGRAM DIRECTORY NOT FOUND

The specified program does not exist or the directory is the wrong one.

DEB205 REGISTER IN THE LEFT PART OF SET IS TOO SHORT

The register specified in the SET command (left of ":=") is too short.

DEB206 ILLEGAL ADDRESS RANGE IN DUMP COMMAND

An address has been specified in the DUMP command that is out of the range of the memory area into which the program has been loaded, or that is within this area but whose offset exceeds the dimensions of the specified segment.

DEB208 IMPLANT TABLE OVERFLOW

More than 19 breakpoints have been set.

DEB209 NOT A BREAKPOINT INTERRUPT AT x..x

No breakpoint has been set at the specified address.
Error emitted by the LIST command.

DEB214 INVALID BRANCH INSTRUCTION

The branch instruction expected at this address cannot be found.

DEB215 VALUE OF RIGHT PART > VARIABLE SIZE

In an assignment instruction the dimensions of the element to be assigned are bigger than those of the variable.

DEB220 NON EXISTENT IMPLANT AT ADDRESS

No breakpoint at this address.

DEB221 PROGRAM SUSPENDED AT: <%xx>%xxxx

The program has been suspended at segment <%xx> with offset %xxxx.

DEB300 EXTENDED INSTRUCTION

DEB301 RESERVED INSTRUCTION

DEB302 PRIVILEGED INSTRUCTION

System instruction; cannot be executed in user mode. The error can only be emitted by an incorrect DUMP command when an odd-numbered offset has been specified.

DEB303 NON EXISTENT INSTRUCTION

An instruction has been input which is not recognised by the debugger (neither privileged nor reserved). Typically a DUMP has been attempted from an area of memory containing instructions specifying an odd-numbered initial instruction.

DEB500 ILLEGAL SYMBOL TYPE IN GETTYPE

DEB501 ILLEGAL VARIABLE TYPE IN GETDIMEN

DEB504 ILLEGAL TYPE IN GETPATENTP

DEB505 ILLEGAL TYPE IN MATCHTYPE

DEB506 ILLEGAL SYMBOL TYPE IN GETNAME

DE8507 ILLEGAL SYMBOL TYPE IN HASH

The ISD is incorrect; it must be reconstructed. If the error continues call the Olivetti Software Maintenance Service.

PR0704 SAINT RETURN STACK OVERFLOW

The identifier input by the user has a syntax which is too complex for the debugger to handle. A simpler identifier must be input. For example, too many subscripts have been nested; the debugger may only handle up to 4.

PART III - GENERALISED UTILITIES

INTRODUCTION TO PART III

This part documents the generalised utilities which service the L1 MOS system.

8. FLOPPY FILE TRANSFER UTILITY (FFT)

INTRODUCTION

FFT is a utility which enables files to be transferred from the S6000 system to the L1 MOS system using floppy disks.

This utility has two versions. One of these runs on the S6000 system and is used to transfer files from the file system to floppy disk. The other version runs on the L1 MOS system and transfers files from floppy disk to the file system.

FILE TYPES

The file types which are supported by the FFT have the following FFT file names:

- BINARY: a binary file is made up of binary data having no logical (e.g. record) structure, e.g. object files created by compilers or translators and the load modules created by the Linker.

- TEXT: a text file contains textual data which is organised in records of variable length, e.g. a source program.

- SEQUENTIAL: a sequential file is made up of records having a fixed length.

- RELATIVE: a relative file is made up of records having a fixed length.

- INDEXED: an indexed file is made up of records of fixed length which may be distinguished via a primary key.

The file types supported by the S6000 and L1 MOS do not have the same type names. The file being converted must first be converted to an FFT file type before it may be transferred to the other system.

The following file conversion table indicates the FFT file type to be declared for each S6000 file which is to be converted to the L1 MOS system.

S6000	FFT	L1 MOS
Positional	Binary	Byte-stream
Positional	Text	Byte-stream
Positional	Sequential	Positional with no record deletion
Keyed	Relative	Positional with record deletion
Keyed	Indexed	Keyed

CHARACTERISTICS AND LIMITATIONS

- A file whose name is the same as an existing file in the set of files handled by the L1 MOS file system may not be transferred. For the successful transfer of this file the existing file must first be deleted.

- 1Mbtye floppy disks must be used to transfer files.

- The FFT cannot handle multi-disk volumes. A user wishing to copy more than one file or a file which is equal to or greater than one Mbyte is advised not to fill any one disk completely.

CALLING THE FFT

The FFT utility is called from the Shell environment by simply entering its name.
FFT call on S6000:

```
HH:MM:SS> fft
```

FFT call on L1 MOS:

```
MCL: fft
```

COMMANDS

The FFT commands are:

- DELETE
- EXIT
- HELP
- INITIALIZE
- LIST
- RESTORE
- SAVE

Only the DELETE, RESTORE and SAVE commands necessitate the use of parameters. Parameters may be entered directly after the command or in interactive mode.

Lower or uppercase letters may be used when entering these commands. It is only necessary to enter the first command letter.

The following is a brief description of each command:

DELETE

Deletes a file previously saved on floppy disk.

D [ELETE] [FileName]

---

where:

- "FileName" is the name of the file to be deleted.

Note

The FFT does not manage file reorganisation and compacting. The disk space recovered from the deleted file may thus be used again only when the deleted file is the last one loaded.

EXIT

Forces the execution of the FFT to end. The system returns to the Shell environment.
The command has no parameters.

HELP

Displays a list of all the FFT commands giving a brief description of each one.

INITIALIZE

Initialises the floppy disk to FFT standards, deleting any data which may be already stored on the disk. The command must be executed before a new disk may be used or a used disk is written to and its old contents deleted.
The command has no parameters.

LIST

Displays the name, type and size of each file stored on the floppy disk. The command has no parameters.

RESTORE

Copies a file stored on floppy disk onto the L1 MOS users working directory. The original file is not deleted.
The command syntax is:

R [ESTORE] [FloppyName [L1Name]]

---

where:

- "FloppyName" is the name of the file stored on floppy disk.
- "L1Name" is the name with which the file is to be stored. If not specified the file will be stored with its original name.

SAVE

Copies a S6000 file onto a floppy disk.
The command syntax is:

S [AVE] [S6000Name [FloppyName]]

---

where:

- "S6000Name" is the name of the file to be copied.
- "FloppyName" is the name with which the file is to be copied onto floppy disk. If omitted the file retains its original name.

Note

The user must enter the file type when the appropriate message appears on screen.

HOW TO USE THE FFT

The following table lists the sequence of operations to be carried out when copying an S6000 file onto the L1 MOS system.

STEP	ACTION ON S6000	RESULT/COMMENT
1	Insert the floppy in drive1.	In most configurations the device name is 10_0.
2	Call the utility. from Shell: HH:MM:SS> fft	The list of available commands is displayed.
3	Initialise the floppy disk via the INITIALIZE command.	This step is mandatory when the floppy disk is new or when the data it contains are to be deleted.
4	Enter the command: SAVE.	The utility requests the name of the file to be transferred and the name with which it is to be saved.
	Enter the name of the file to be transferred and its new name.	The utility requests the name of the file type.
	Enter the file type.	The file is transferred onto floppy disk.
5	Enter the command: EXIT	The system returns to the Shell environment.
6	Remove the floppy disk from drive1.	

STEP	ACTION ON L1 MOS	RESULT / COMMENT
7	Insert the floppy disk on drive1 of the L1 machine.	The device name is DEV/FL1.
8	Call the utility from Shell: MCL: fft	The list of available commands is displayed.
9	Enter the command: LIST	The list of files stored on the floppy disk is displayed (this command is optional).
10	Enter the command: RESTORE	The utility requests the name of the file to be transferred and the name with which it to be stored.
	Enter the name of the file to be transfer and its new name.	The file is stored under the working directory.
11	Enter the command: EXIT	The system returns to the Shell environment.

NOTES

The following are some notes which may be of interest to the user. They concern :

- the floppy disk format after it has been initialised
- the transfer of files which have secondary keys.

FORMAT OF AN FFT FLOPPY DISK

The floppy disk format is given in the table below:

HEADER BLOCK:

Address	Description
0	8 byte string containing the floppy disk identifier ('FFT V1.2').
8	16 bit integer containing the header length in bytes.
10	16 bit integer containing the number of files saved.
12+48*(N-1)	32 bit integer containing the address of the nth file.
16+48*(N-1)	32 bit integer containing the length of the nth file.
20+48*(N-1)	32 bit string containing the name of the nth file (must terminate with "ii").
52+48*(N-1)	16 bit code indicating the name of the nth file.
54+48*(N-1)	16 bit integer indicating the record length in bytes.
56+48*(N-1)	16 bit integer indicating the offset of the key within th record.
58+48*(N-1)	16 bit integer indicating the length in bytes of the key within the record.

DATA BLOCKS:

ADDRESS	DESCRIPTION
4096	Data of transferred files.

where:

- The header is a memory area of 4096 bytes which may contain information on a maximum of 85 files stored on disk.
- The size of the internal buffer for the transfer of data is 4096 bytes, records must thus be less than 4096 bytes in length. The records of a positional file must be less than 4092 bytes since 4 bytes are reserved for the record key.
- Textual file data includes the "linefeed" character which separates the records.

TRANSFERRING FILES WITH SECONDARY KEYS

The FFT utility does not handle files with secondary keys. These files must thus be redefined before they may be transferred to the L1 MOS system.

Example:

The debug file "filename_DB" with secondary keys is to be transferred to the L1 MOS system.
The file records have the following keys:

- a primary key of 4 bytes with offset 0.
- a secondary key of 30 bytes with offset 5.

The following command must be specified in order to redefine the secondary keys which lose their meaning when the file is transferred.

```
MCL:    MKINDEX filename_DB D 5 30
```

where:

- "filename-DB" is the name of the debug file.
- "0" specifies that the secondary key may be duplicated.
- "5" indicates that the offset of the secondary key is 5 bytes.
- "30" indicates that the length of the secondary key is 30 bytes.

9. UTILITY FOR PRODUCING STATISTICS ABOUT OBJECT MODULES (CHECK)

The CHECK utility produces statistical information about specified object modules, and executes a syntax analysis on an OLINK command file. These object modules are output from the COBOL, BASIC, FORTRAN and Pascal+ compilers.

This utility provides information that can be extremely useful when designing large overlay programs (see Chapter 1 - Characteristics and Limitations). Whenever the limits on sections, symbols, or references (these are most frequent) are exceeded, the source program must be modified, as advised in the Program Preparation manual for the language being used.

The following information is output for each module:

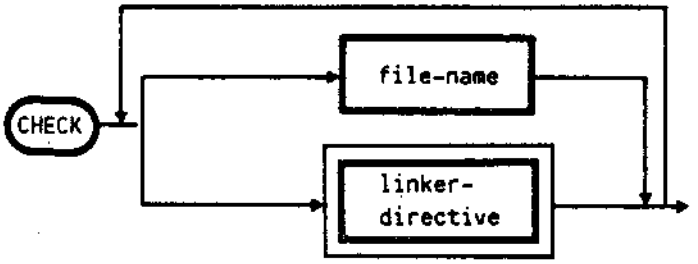
- name of file containing the module
- name of sections and their size
- number of sections
- number of symbols defined
- number of references.

A review containing the following is then produced:

- the total number of sections
- the total number of symbols
- the total number of references and symbols
- the total number of undefined symbols.

CALLING A CHECK

The syntax for the CHECK call is:



where:

file-name specifies the name of file input to CHECK, and can be one or more of the following:

- an object file
- an object file from a library
- a load-module
- a linker command file
- a file of symbols

The file is identified by a path-name that adopts the same conventions used in Shell environment.

linker-directive represents the linker commands described in Chapter 3.

The following option is assumed by default:

OPTIONS SILENT

If the PROGRAM directive is missing, a program directory called CHECK is created in the same way as OLINK.

If a library is specified in input, CHECK provides the statistical data about the objects taken from the library, using the undefined symbols present when the library is processed.

For error messages issued by the utility CHECK see Chapter 4 in this manual.

EXAMPLE OF THE OUTPUT PRODUCED BY CHECK

```
- CHECK 7.0 -        - MOS REL. 5.1 -        DIAGNOSTICS

------------ COMMAND LANGUAGE INTERPRETATION ---------------
ARG: linkcmd
<linkcmd>: COMMAND FILE:
     CMD: options no silent
     CMD:              map linkmap
     CMD:             output linkout
     CMD:             entry Pmain
     CMD:              Optimize
     CMD:              input
     CMD:                   driver.obj
     <driver.obj>: OBJECT FILE
          SECTION NAME    PROG_p      SIZE    112
          SECTION NAME    PROG_d      SIZE    10
          SECTION NAME    PROG_s      SIZE    0
          SECTION NAME    PROG_k      SIZE    34
          SECTIONS              4
          DEFINED SYMBOLS       4
          REFERENCES            7
     CMD:                   m_main.obj
     <m_main.obj>: OBJECT FILE
          SECTION NAME    m_main_p      SIZE    3187
          SECTION NAME    m_main_d      SIZE    12
          SECTION NAME    m_main_s      SIZE    0
          SECTION NAME    m_main_k      SIZE    438
          SECTIONS              4
          DEFINED SYMBOLS       5
          REFERENCES            43
     CMD:                   m_outout.obj
     <m_output.obj>: OBJECT FILE
          SECTION NAME    m_output_p      SIZE    1146
          SECTION NAME    m_output_d      SIZE    22
          SECTION NAME    m_output_s      SIZE    0
          SECTION NAME    m_output_k      SIZE    0
          SECTIONS              4
          DEFINED SYMBOLS       10
          REFERENCES            6
     CMD:                   m_process.obj
     <m_process.obj>: OBJECT FILE
          SECTION NAME    m_process_p      SIZE    2E20
          SECTION NAME    m_process_d      SIZE    560
          SECTION NAME    m_process_s      SIZE    0
          SECTION NAME    m_process_k      SIZE    114
          SECTIONS              4
          DEFINED SYMBOLS       20
          REFERENCES            20
     CMD:                   m_utility.obj
     <m_utility.obj>: OBJECT FILE
          SECTION NAME    m_utility_p      SIZE    390
          SECTION NAME    m_utility_d      SIZE    0
          SECTION NAME    m_utility_s      SIZE    0
          SECTION NAME    m_utility_k      SIZE    36
          SECTIONS              4
          DEFINED SYMBOLS       1
          REFERENCES            10
     CMD:                   m_arg.obj
     <m_arg.obj>: OBJECT FILE
          SECTION NAME    m_getarg_p      SIZE    366
          SECTION NAME    m_getarg_d      SIZE    0
          SECTION NAME    m_getarg_s      SIZE    0
          SECTION NAME    m_getarg_k      SIZE    40
          SECTIONS              4
          DEFINED SYMBOLS       2
```

```
- CHECK 7.0 -      - MOS REL. 5.1 -      DIAGNOSTICS
        REFERENCES              17
    CMD:        /usr/olivetti/libD30/interface/LPRT.obj
    </usr/olivetti/libD30/interface/LPRT.obj>: OBJECT FILE
        SECTION NAME    stdenv_p      SIZE    732
        SECTION NAME    stdenv_d      SIZE    2
        SECTION NAME    stdenv_s      SIZE    0
        SECTION NAME    stdenv_k      SIZE    56
        SECTIONS                4
        DEFINED SYMBOLS         7
        REFERENCES              13
        SECTION NAME    zio_p       SIZE    0
        SECTION NAME    zio_d       SIZE    1092
        SECTION NAME    zio_s       SIZE    0
        SECTION NAME    zio_k       SIZE    0
        SECTIONS                4
        DEFINED SYMBOLS         2
        REFERENCES              3
        SECTION NAME    zre_p       SIZE    12
        SECTION NAME    zre_d       SIZE    0
        SECTIONS                2
        DEFINED SYMBOLS         1
        REFERENCES              0
    CMD:            type 2   attributes 1   <<45>>0000 PROG_p *_[pP]
    CMD:            type 5   attributes 0   <<4e>>0000 *_[dD] *_k
    CMD:            type 6   attributes 32  <<63>>0000 *_[sS]
    CMD:            type 5   attributes 0   <<61>>0000 *_i <<61>>%2000 *_e
    CMD:
    END OF COMMAND FILE "linkcmd"
TOTAL SECTIONS              34
TOTAL SYMBOLS               83
TOTAL REFERENCES            121
TOTAL UNDEFINED SYMBOLS     25
```

10. UTILITY FOR DUMPING A MOS L-MODULE (M5LDUMP)

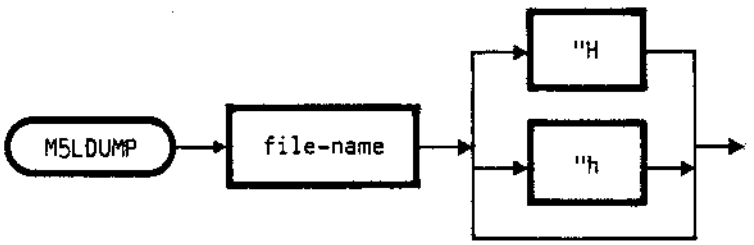
The M5LDUMP utility reads a MOS l-module (see Chapter 2, "L-Module Format" Section in the MOS Programmer Guide), identified by the pathname, and the following information is output:

- The length and header of the l-module
- The DATA SECTION headers contained in every segment followed by the "data" part containing the related data/code; at the linking phase it is possible to insert in the same segment sections having different contents.
- Additional information present in the file

The output produced is partly in symbolic format and partly in hexadecimal format.

M5LDUMP CALL

The utility call syntax is as follows:



where:

file-name identifies the pathname of the l-module. It may be expressed either as an absolute pathname, or as a pathname relative to the working directory of the station that activated the command.

"H or **"h** is an option that limits the display of the l-module to the DATA SECTION header, without the "data" part.

When M5LDUMP is activated it checks that the pathname specified relates to an l-module: otherwise it sends a warning to the user.

EXAMPLE OF OUTPUT FROM M5LDUMP

```
 OLIVETTI M5LDUMP -- REL 0.1
  DUMPING FILE : SHDATE

<LENGTH>                  4074
<HEADER>
    <HEADER_COUNT>        71
    <MESSAGE>
        <MSGCOUNT>        23
        <MSGDATA>         .. -SHDATE-D60-Jun11-19
        <EOMBYTE>         ##FF

    <ENTRY_POINT> :       <SEGNUM>  = 51     <OFFSET> = 0
    <SEG_RANGE>   :       <SEG_LOW> = 50     <SEG_UP>  = 63


    <SEG_INFO_TABLE>
       <SIT_COUNT>    = 3
       <SIT_ENTRY> ::= <SEGNUM> <SEGLENGTH> <SEGATTR> <SEGTYPE> <DATAPOINTER>:
            0              50          6         0          5             75
            1              51         15         1          2            294
            2              63          0        32          6              0
    <EOHBYTE> = ##FF


<DATA SECTION> at 75
    <DATA_POINTER>    = 0
    <SEC_COUNT>       = 204
    <SEG_NUM>         = 50
    <OFFSET>          = 1158
    <EOSBYTE>         = ##FF

0000 0000 0000 00CC  0032 0000 0486 0000
0000 0000 0000 4000  0000 4000 0000 0000
0000 0000 0000 0000  0000 0000 0000 2049
4E56 414C 4944 204F  5054 494F 4E00 1F1C
1F1E 1F1E 1F1F 1E1F  1E1F 1F10 1F1E 1F1E
1F1F 1E1F 1E1F 00BA  00C6 00D2 00DE 00EA
00F6 0102 010E 013A  0146 0152 015E 016A
0176 0182 018E 019A  01A6 01B2 0280 028C
            .                 .
            .                 .
            .                 .
```

(cont.)

```
                .                   .
                .                   .
                .                   .

6572 726F 7220 2D20  5072 6F67 7261 6D20
7465 726D 696E 6174  6564 7374

<DATA SECTION> at 294
    <DATA_POINTER>    = 0
    <SEC_COUNT>       = 3764
    <SEG_NUM>         = 51
    <OFFSET>          = 0
    <EOSBYTE>         = ##FF

0000 0000 0000 0E84  0033 0000 0000 5E08
8300 0006 030F 005C  0DE5 005C 5C09 0203
8200 0000 5F00 8300  0A8E 4005 8200 0008
007F 5F00 8300 0D24  5D06 8200 0012 5402
8200 0012 7604 8200  0016 5F00 8300 0108
7602 8200 0016 7604  8200 0022 5F00 8300
05DE 140C 246F 7074  5D0C 8200 00CA 5402
8200 0000 6104 8200  0004 6105 8200 0006

                .                   .
                .                   .
                .                   .

5E08 0900 1408 0078  0000 5E08 0900 1408
0078 0008 5E08 0900  1408 0078 0050 5E08
0900 1408 0078 00E0  5E08 0900 1408 0078
0110 5E08 0900 1408  0078 0170 5E08 0900
1408 0078 01B0 5E08  0900 1408 0078 0188
5E08 0900 1408 0078  01C0 5E08 0900 1408
0078 01C8 5E08 0900  1408 0078 01D0 5E08
0900 1408 0078 01D8  5E08 0900 1408 0078

01E0 5E08 0900 1408  0078 01E8 5E08 0900
1408 0078 01F0 5E08  0900 1408 0078 01F8
5E08 0900 1408 0078  0200 5E08 0900 1408
0078 0208

    <EOFBYTE> = FF

< ADDITIONAL INFORMATION FROM POS 4073 TO POS 4095  >
FF00 0000 0000 0000  0000 0000 0000 0000
0000 0000 0000

<<                      MSLDUMP COMPLETE                        >>
```

A. DEBUGGER MODULES

The symbolic debugger comprises three modules:

```
MAIN
err.d
bt/MAIN
```

where:

MAIN is the actual debugger

err.d is the error message file

bt/MAIN is the ISD controller (Internal Symbol Dictionary).

If the modules are not installed under the same program-directory an error message is output and the debugger terminates.

Module names are fixed.

The user program must be loaded using the name MAIN under the program-directory, as for ISD, if this exists, and for any overlays that may exist.

Example

```
Debugger program-directory:     DEB

                   modules:     DEB/MAIN
                                DEB/bt/MAIN
                                DEB/err.d

User program-directory:         PRISCILLA

                 modules:       PRISCILLA/MAIN
                                PRISCILLA/ISD      (if it exists)
                                PRISCILLA/overlay1 (if it exists)
                                        .
                                        .
                                        .
                                PRISCILLA/overlayN (if it exists)
```

B. TABLE SHOWING THE GRAPHIC EQUIVALENCE OF ASCII CHARACTERS

The following table indicates, for each country, the graphic equivalent of each ASCII characters specified.

ASCII VALUE		NATIONAL EQUIVALENT														
DECIMAL	HEXADECIMAL	USA	ITALY	FRANCE	GREAT BRITAIN	GERMANY (ORIGINAL)	GERMANY (WEST)	SPAIN	PORTUGAL	DENMARK	SWEDEN FINLAND	NORWAY	SWITZERLAND FRENCH	SWITZERLAND GERMAN	GREECE	YUGOSLAVIA
35	23	(#)	£	£	£	#	#	£	#	£	#	£	£	£	£	#
36	24	($)	$	$	$	$	$	$	$	$	¤	$	$	$	$	¤
64	40	(@)	§	à	@	§	§	§	§	·	@	·	§	§	·	Ś
91	5B	([)	°	°	[	Ä	Ä	¡	Ã	Æ	Ä	Æ	à	à	[	đ
92	5C	\	ç	ç	\	Ö	Ö	Ñ	Ç	Ø	Ö	Ø	ç	ç	\	Ć
93	5D	(])	é	§	]	Ü	Ü	c	Õ	Å	Å	Å	è	è	]	Ź
96	60	`	ù	`	`	`	`	`	`	`	`	`	`	`	`	ś
123	7B	({)	à	é	{	a	a	°	ã	æ	ä	æ	ä	ä	{	đ
124	7C	(|)	ò	ù	|	o	ö	ñ	ç	ø	ö	ø	o	ö	|	ć
125	7D	(})	è	è	}	ü	ü	ç	õ	å	å	å	ü	ü	}	ź
126	7E	(~)	ì	¨	~	ß	ß	~	°	¨	—	¨	ė	é		č

* Encircled characters are used in debug commands.