UNREL(TM) - REL to ASM translator

Table of Contents

Published by MISOSYS, Inc., PO BOX 239, Sterling, VA 22170-0239.

General

This product can be used to aid in the conversion of relocatable object modules to a Z80 source code file. It consists of three utilities: UNREL, DECODREL, and SPLLTLIB. UNREL will decode a relocatable object module which has been assembled by either Microsoft's M80 or MISOSYS' MRAS assemblers. The output is an assembler source file compatible with MRAS and one which should also be equally usable with M80. DECODREL can be used to obtain a bit-stream analysis of a REL module or library. Finally, SPLITLIB can be used to break a large REL library into smaller pieces suitable for loading into memory constrained REL Librarians (so that your librarian can extract single modules to be UNRELed).

This documentation provides information on both the UNREL-T80 product (usable on a TRS-80 with either LDOS(TM) 5.x or LS-DOS(TM) 6.x) and the UNREL-CP/M product (usable with CP/M 2.x or 3.x). Although file specifications are shown in this document in the form, filename/ext:d under CP/M this will be assumed to represent the form, d:filename.ext The CP/M version requires a Z80 based computer.

UNREL(TM) - REL to ASM translator

The files on the accompanying DATA diskette may be easily copied to your working SYSTEM disk by means of the DOS COPY (or PIP) utility. There may be a file named "README/TXT" on the disk. If so, that file will contain important information which may not appear in this printed documentation. You should read this file by issuing the command: LIST README or TYPE README.TXT.

UNREL - REL to ASM translator

UNREL will decode a relocatable object module which has been assembled by either Microsoft's M80 or MISOSYS' MRAS assemblers. The output is an ASCII assembler source file using Z80 mnemonics. Invoke UNREL with the syntax:

```
 ____________________________________________________
|                                                    |
| UNREL infile[/REL] [outfile[/ASM]]                 |
|                                                    |
| infile  - Is the filename of the REL module. If    |
|           the extension is omitted, 'REL' will     |
|           be assumed.                              |
|                                                    |
| outfile - Is the name to be used for the output    |
|           Z80 assembler file. If omitted, then     |
|           "infile/ASM" will be used. If outfile    |
|           is entered without an extension, 'ASM'   |
|           will be assumed. The source drive will   |
|           be used unless outfile includes a        |
|           drive specification.                     |
|____________________________________________________|
```

The action of UNREL is to take a binary REL file which looks like this when displayed in hexadecimal:

```
8091D15391D49420450534554819414915 4D155205504F494E5481131253
```

into a form more usable by your MRAS or M80 assembler; an ASCII file such as the following:

```
;GENGRP/ASM:1
NAME   ('GENGRP')
EXTRN   BAKCLR,CLIPP,DCOMPR,DOWNC,FCERR,FETCHC,FORCLR,GRPACX
EXTRN   GKPACY,GXPOS,GYPOS,ICOMP,LEFTC,LINSTL,MAPXYC,MAXDEL
EXTRN   MAXUPD,MINDEL,MINUPD,NSETCX,PGRPCX,PGRPC,READC,RIGHTC
EXTRN   SCALXY,SCNCRD,SEEGRP,SETATR,SETC,STOREC,UPC,VIEWMP
EXTRN   VXLEFT,VXRGHT,VYLEFT,VYRGHT,XCHGAC,XCHGX,XCHGY
PUBLIC  BOXLLN,CLS,DDT,DOBOXF,DOGRPH,DOLINE,GETPXY
PUBLIC  GRPCL1,HLFDE,LINE,LINEB,LINEBF,NEGHL,POINT
PUBLIC  PSET,SETXY,SETXYR,VIEW,XDELT,YDELT
CSEG
SETXYR:
PUSH      DE
PUSH      HL
LD   HL,(GRPACX)
LD   (PGRPCX),HL
```

```
     EX   (SP),HL
     LD   E,(HL)
     INC  HL
     LD   D,(HL)
     POP  HL
     ADD  HL,DE
     LD   (GRPACX),HL
     LD   (GXPOS),HL
     LD   HL,(GRPACY)
     LD   (PCRPCY),HL
     EX   (SP),HL
     LD   E,(HL)
     INC  HL
     LD   D,(HL)
     POP  HL
     ADD  HL,DE
     LD   (GRPACY),HL
     LD   (GYPOS),HL
     RET
SETXY:
```

The example above, incidentally, is from the graphics library, GRPLIB/REL, which is distributed with Tandy's hi res graphics board. This miraculous transformation is made possible by the UNREL utility. Of course, there are limitations.

First, UNREL makes the assumptions that anything in a code segment is code, and anything in a data segment is data. Those of you already having experience with object code disassemblers know that decoders can sometimes get "fooled" by data being interpreted as code. With object module REL files, this problem still exists. However, if good programming practices, such as segregation of code and data, have been followed by the original programmer of the REL module, your decoding job is simpler.

Second, UNREL supports only the following special link items (SLI):

```
 O - entry symbol
 1 - select common block
 2 - program name
 3 - request library search
 5 - define common size
 6 - chain external
 7 - define entry point
 9 - extern+offset
10 - define data size
11 - set location counter
13 - define program size
14 - end program
15 - end file
```

The undocumented special link items (4 and 8) as well as chain address, SLI-12, are not supported. The later is used in Microsoft's one-pass compilers. SLI-4 and SLI-8 are used in a more recent version of M80 for handling 8-bit externs with greater support of arithmetic expressions resolvable at link time; however, Microsoft wasn't too potent in letting the world know of the specific details associated with that link item. In any event, we're not supporting it in UNKEL.

UNREL(TM) - REL to ASM translator

Here is a small sample program used to demonstrate the behavior of UNREL. It is the assembly listing using MRAS.

```
mras testmod:l

MRAS 1.0a - Copyright (c) 1985 MTSOSYS, Inc., All rights reserved.

Including TESTMOD:1
               00001          CSEG
0000' 210000"  00002 START    LD        HL,MESSAGE
0003' 3EOA     00003          LD        A,10
0005' EF       00004          RST       28H
0006' C30000*  00005          JP        NEXT##
               00006          DSEG
0000" 48       00007 MESSAGE DB        `Hello World!',13
     65 6C 6C 6F 20 77 6F 72
     6C 64 21 OD
0000'          00008          END       START
0000 is the transfer address
00000 Total errors
36974 Free space
```

Note that this module contains an external symbol, "NEXT". It also contains both a code and a data segment. There are also two symbols local to the module; one in each segment. We can process the resulting REL module with UNREL using a command such as:

```
UNREL TESTMOD
```

UNREL will generate the file, TESTMOD/ASM, which contains the assembler source code for "testmod" as determined from its relocatable object module. During the translation process, UNREL will display some messages which indicate its progress. These messages will look like the following:

```
UNREL - Disassemble /REL Module - Version 1.0b
Copyright 1987 Riclin Computer Products. All rights reserved.

Pass 1 - building segments and symbol tables
..................................................
Pass 2 - resolving relocations and local labels
....
Pass 3 - disassembling to output
.................
```

After this operation, UNREL has generated the source file which can then be assembled by either M80 or MRAS (or other M80-compatible assembler). The "assembler source" file would look like this:

```
;TESTMOD/ASM:1
     NAME           (`TESTMO')
     EXTRN     NEXT
     CSEC,
CSEG$0000:
     LD        HL,DSEG$0000
     LD        A,0AH
     RST       28H
     JP        NEXT
     DSEG
```

```
    DSEGS0000:
         DB   48H
         DB   65H
         DB   6CH
         DB   6CH
         DB   6FH
         DB   20H
         DB   77H
         DB   6FH
         DB   72H
         DB   6CH
         DB   64H
         DB   21H
         DB   0DH
         END  CSEG$0000
```

As can be observed, the external symbol has been noted by means of an "EXTRN" statement. The source code prologue section will include all such externals and PUBLIC symbols as well. Symbols local to the module will be identified by a unique symbol. The data segment region will be generated as discrete byte values.

In cases where the original assembler source module was fraught with non-code fragments in the code segment or code fragments in the data or common segments, UNREL will not be able to generate an "accurate" representation of the original code. This is to be expected. Also, UNREL will group all code segments into one single code segment. Similarly, all data segments will be grouped into one. This does not effect the logic of the original source code.

UNREL can work properly only on a single module; don't expect it to produce meaningful output if you try to "unrel". a library composed of more than one module. If you have the MISOSYS librarian for M-80 type REL files, MLIB, or a CP/M librarian, such as LIB80 or LIB, you can easily pull apart relocatable libraries into their individual module members and then translate the associated modules into ASM source. Without the resources of a librarian, you may be able to split a library into single modules with the SPLITLIB utility provided as part of this package.

UNREL(TM) - REL to ASM translator

DECODREL - REL bit stream analysis

The DECODREL utility generates an analysis of the bit stream of a REL file. This can be used to more fully understand the actual bit stream. DECODREL is invoked with the syntax:

```
 ______________________________________________________
| DECODREL [-f] infile[/REL] [outfile[/RMP]]           |
|                                                      |
| -f      - Flag used to designate a FULL output       |
|           versus a brief output. DECODREL will       |
|           default to brief. Specify "-f" for a       |
|           FULL report.                               |
|                                                      |
| infile  - Is the filename of the REL module. If      |
|           the extension is omitted, 'REL' will       |
|           be assumed.                                |
|                                                      |
| outfile - Is the name to be used for the output      |
|           analysis file. If omitted, Then            |
|           "infile/RMP" will be used. If outfile      |
|           is entered without an extension, 'RMP'     |
|           will be assumed. The source drive will     |
|           be used unless outfile includes a          |
|           drive specification.                       |
|______________________________________________________|
```

If we want to process the TESTMOD/REL module previously illustrated, and generate an analysis map, we would use a command such as:

```
DECODREL TESTMOD
```

The following analysis is generated to the file named, "TESTMOD/RMP":

```
0000:7 - Program name (02), TESTMO
0007:5 - Data area size (OA), Absolute (O), 000D
0OOA:4 - Program size (OD), Program relative (1), 0009
000D:3 - Set counter (OB), Program relative (1), 0000
001A:0 - Set counter (OB), Data relative (2), 0000
002C:2 - Chain external (06), Program relative (1), 0007, NEXT
0034:6 - End program (OE), Program relative (1), 0000
0038:7 - End file (OF)
```

Since we did not specify the "-F" flag, the BRIEF analysis is generated. Such an analysis does not include any absolute, data relative, code relative, or common relative bytes. The presentation includes the bit-stream flow of special link items.

The first field noted is the relative byte and bit offset of the referenced item within the relocatable object module file (remember, bits are used hi-order to lo-order or 7 to 0). This may prove useful for specialized purposes. The information presented on each line describes the special link item by name; the contents of (and description where appropriate) each SLI field is also noted in hexadecimal. The technical specification section describes the structure of an M80-compatible relocatable object module. That section should be reviewed if you are either unfamiliar with that format or have, at most, read only sketchy details of the format.

If you would prefer the DECODREL analysis to include information on the entire REL module bit stream, the preceding command invocation would be changed to:

DECODREL -F TESTMOD ALLDATA

This example also illustrates the specification of "outfile". The following analysis is generated to the file named, "ALLDATA/RMP":

```
0000:7 - Program name (02), TESTMO
0007:5 - Data area size (0A), Absolute (O), 000D
000A:4 - Program size (0D), Program relative (1), 0009
000D:3 - Set counter (0B), Program relative (1), 0000
0010:2 - 0000' - Absolute item, 21
0011:1 - 0001' - Data relative (2), 0000
0014:6 - 0003' - Absolute item, 3E
0015:5 - 0004' - Absolute item, 0A
0016:4 - 0005' - Absolute item, EF
0017:3 - 0006' - Absolute item, C3
0018:2 - 0007' - Absolute item, 00
0019:1 - 0008' - Absolute item, 00
001A:0 - Set counter (0B), Data relative 92), 0000
001E:7 - 0000" - Absolute item, 48
OO1F:6 - 0001" - Absolute item, 65
0020:5 - 0002" - Absolute item, 6C
0021:4 - 0003" - Absolute item, 6C
0022:3 - 0004" - Absolute item, 6F
0023:2 - 0005" - Absolute item, 20
0024:1 - 0006" - Absolute item, 77
0025:0 - 0007" - Absolute item, 6F
0027:7 - 0008" - Absolute item, 72
0028:6 - 0009" - Absolute item, 6C
0029:5 - 000A" - Absolute item, 64
002A:4 - 000B" - Absolute item, 21
002B:3 - 000C" - Absolute item, 0D
002C:2 - Chain external (06), Program relative (1), 0007, NEXT
0034:6 - End program (OE), Program relative (1), 0000
0038:7 - End file (0F)
```

This analysis includes the segment relative address of each item being presented followed by the standard segment indicator character.

SPLITLIB - REL library splitter

Librarians which work by loading a REL library into a memory buffer may limit the size of the library it can deal with. This is the case with MLIB. ,To overcome this limitation, SPLITLIB can be used to split a large library file into two or more smaller files. SPLITLIB is invoked with the syntax:

```
 ______________________________________________________
|                                                      |
| SPLITLIB infile[/REL] maxlength [drivespec]          |
|                                                      |
|                                                      |
| infile     - Is the filename of the REL library. If  |
|              the extension is omitted, 'REL' will    |
|              be assumed.                             |
|                                                      |
| maxlength  - Is the maximum length of an output      |
|              file (in bytes). The module currently   |
|              being output will be continued to it's  |
|              "module end" which will be followed by  |
|              an "end file" byte (X'9E').Maxlength    |
|              must be in the range <100-32767).       |
|                                                      |
| drivespec  - This designates the drive to which the  |
|              file partitions will be written. If     |
|              omitted, the drive specified with       |
|              "infile" will be used. Each output      |
|              partition will be named, "infile/Rxx";  |
|              "xx" being 01, 02, ... for the first,   |
|              second, etc., partitions.               |
|______________________________________________________|
```

Let's say we have a library named GRPLIB that we wish to split. The following example illustrates splitting this library file into pieces approximately 6000 bytes in length:

```
slibd grplib:7 6000 :1

SPLITLIB - Split /REL Library - Version 1.0a
Copyright 1986 Riclin Computer Products. All rights reserved.

Reading input file GRPLIB/REL:7

Writing output file GRPLIB/RO1:1
Module ADVGRP
Module GENGRP

Writing output file GRPLIB/R02:1
Module TRSGRP
```

This example illustrates how SPLITLIB informs you of the file it is reading, the output files being generated, and the modules being written to each output file partition.

If the last module of the source file being written to the last output partition results in a partition size exceeding maxlength, another file of NULL length will be generated. This NULL file can be ignored.

UNREL(TM) - REL to ASM translator

Technical specifications

This section describes the relocatable bit stream of a Microsoftrelocatable object module. We do not intend this section to be an authority on the subject; however, its discussion accurately portrays our interpretation of the documentation appearing in the literature presented by Microsoft.

Microsoft compatible 'REL' format

All Z80 assemblers work in a similar fashion, in that they convert a file containing SOURCE CODE, written in Z80 assembly language mnemonics, to OBJECT CODE in some binary format. In ABSOLUTE assemblers, this binary data is a faithful representation of the actual machine language (ones and zeros) that the Z80 will execute when you want your program to run. This object code can only load and execute at a FIXED address in the Z80's memory space. On the other hand, a RELOCATABLE assembler, such as M80 or MRAS, will generate object code which can be relocated to any address in the Z80's 64K memory space before the program is executed.

Let's look at an example of absolute assembly. The following program has been assembled at an ORIGIN of 0100H. Notice especially the values assigned to the memory addresses @DATE, @EXIT, @DSPLY, START, and BUFFER:

```
0100            00100          ORG  0100H
4470      00110 @DATE    EQU  4470H
402D            00120 @EXIT    EQU  402DH
4467            00130 @DSPLY   EQU  4467H
000D            00140 CR       EQU  ODH
0100 211401     00150 START:   LD   HL,BUFFER
0103 CD7044     00160          CALL @DATE
0106 3E0D       00170          LD   A,CR
0108 321C01     00180          LD   (BUFFER+8),A
010B 211401     00190          LD   HL,BUFFER
010E CD6744     00200          CALL @DSPLY
0111 C32D40     00210          JP   @EXIT
0114            00220 BUFFER:  DS   9
0100            00230          END  START

@DATE       4470 @DSPLY        4467 @EXIT         402D
BUFFER      0114 CR            000D START         0100
```

The program has been reassembled below at a new origin, 0200H. Some of the addresses for the above labels have changed, while some remain the same:

```
0200            00100          ORG  0200H
4470            00110 @DATE    EQU  4470H
402D            00120 @EXIT    EQU  402DH
4467            00130 @DSPLY   EQU  4467H
000D            00140 CR       EQU  ODH
0200 211402     00150 START:   LD   HL,BUFFER
0203 CD7044     00160          CALL @DATE
0206 3E0D       00170          LD   A,CR
0208 321C02     00180          LD   (BUFFER+8),A
020B 211402     00190          LD   HL,BUFFER
020E CD6744     00200          CALL @DSPLY
0211 C32D40     00210          JP   @EXIT
0214            00220 BUFFER:  DS   9
```

```
     0200           00230          END  START

     @DATE       4470 @DSPLY        4467 @EXIT         402D
     BUFFER      0214 CR            000D START         0200
```

To be specific, START and BUFFER have changed, while the others are unchanged. Both START and BUFFER have been relocated! START, instead of being at 0100H is now at 0200H, and BUFFER has moved from 0114H to 0214H. This offset of 0100H is due to the changed origin, 0100H versus 0200H. START and BUFFER are therefore internally relocatable values, while @DATE, for example, will always be 4470B, and is thus known as an absolute value.

The same program, as assembled using relocation looks like this:

```
     4470                 @DATE   EQU  4470H
     402D                 @EXIT   EQU  402DH
     4467                 @DSPLY  EQU  4467H
     000D                 CR      EQU  0DH
     0000' 21 0014'       START:  LD   HL,BUFFER
     0003' CD 4470                CALL @DATE
     0006' 3E 0D                  LD   A,CR
     0008' 32 001C'               LD   (BUFFER+8),A
     000B' 21 0014'               LD   HL,BUFFER
     000E' CD 4467                CALL @DSPLY
     0011' C3 402D                JP   @EXIT
     0014'                BUFFER: DS   9
                                  END  START

     @DATE       4470  @DSPLY       4467 @EXIT         402D
     BUFFER      0014' CR           000D START         0000'
```

All of the internal program addresses have been assembled as if the program had an origin of 0000H, and are noted with a following single-quote ('). This is relocation at work. The binary output of this assembly (a /REL file) cannot be executed by the Z80 until you choose an origin for the program; this is done by a utility known as a LINKER, and can be ANY address in the Z80 memory space. The linker will determine, from the origin you have selected, where START and BUFFER really will be when the program is run. If you choose 0100H as the origin, then START will be located at 0100H, and BUFFER at 0114H. Other origins will produce similar results; START and BUFFER will be at different addresses, but the offset between them (0014H) will always be the same.

This characteristic of relocatable object files, that they can be LINKED at any origin, is extended by a further capability: relocatable object files may be linked TOGETHER to form a complete program from many smaller pieces. This allows you to write a very large program in lesser chunks which are easier to edit and to understand. In addition, you can develop libraries of standard and useful subroutines, each thoroughly tested and debugged, which any main program may call upon when necessary. The Microsoft FORTRAN library (FORLIB/REL), for example, thus contains many subroutines which can be used by any FORTRAN or Z80 assembler program.

The mechanism of program and subroutine linkage that is often used is implemented by the ENTRY and EXTERNAL attributes. A label which is declared ENTRY (or GLOBAL or PUBLIC) in one module can be accessed by another module in the following way:

```
;Module 1
     ENTRY   LABEL1      ;this is an entry point
 LABEL1:
     <code follows>
     END                 ;end of module 1

;Module 2
     EXTRN   LABEL1      ;this is an EXTERNAL declaration
                         ;could also be EXT.
     CALL    LABEL1      ;and this is a reference to the
                         ;external
     END                 ;end of module 2
```

The relocatable format also allows you to do other things. In many applications, program code and data areas must be separated. This most often occurs when code must be placed in ROM, such as the BASIC interpreter in a TRS-80. However, the data areas cannot be in ROM; they must be inwriteable memory (RAM), and thus must be separated from the code areas. This can be accomplished by use of the CSEG and DSEG commands to the assembler. A CSEG pseudo-operation signals the start of a code area, while a DSEG indicates the start of a data area. Code and data SEGMENTS may be intermixed in a program source file, and the assembler will automatically keep them separate by the use of two distinct program or location counters, one for each segment. When you link a program with the linker, you may tell the linker at what address to place the code, and also where to place the data. Thus the two segments are separated. The above example is shown below using this technique:

```
4470                    @DATE    EQU    4470H
402D                    @EXIT    EQU    402DH
4467                       @DSPLY   EQU    4467H
000D                       CR       EQU    0DH
0000'                          CSEG   ;code starts here
0000'     21 0000"         START:   LD     HL,BUFFER
0003'   CD 4470                     CALL   @DATE
0006'   3E OD                  LD     (BUFFER+8),A
0008'   32 0008"                    LD     HL,BUFFER
000B'   21 0000"                    CALL   @DSPLY
000E'   CD 4467                     JP     @EXIT
0011'   C3 402D                     DSEG   ;data starts here
                           BUFFER:  DS     9
0000"                               END    START


@DATE        4470  @DSPLY       4467   @EXIT     402D
BUFFER       0000" CR           000D   START     0000'
```

Notice how the label BUFFER is now located at 0000H, but in the data segment, as indicated by the double-quote (") following the address. An linker session (illustrated with MLINK commands) could then be as follows with user entries in BOLDFACE:

```
DOS Ready
MLINK
MLINK - Ver 1.0a Copyright 1985 MISOSYS, Inc., All rights reserved
?-p=100
?-d=1000
?test
27937 Free space
```

```
P <0100-0113 0014> D <1000-1008 0009>
*test-n-e
DOW Ready
```

The "-p" command to the linker established the program (or code) segment origin, while the "-d" command did the same for the data segment. After loading TEST/REL with the next command, the linker then tells us where the two segments are located and how long they are. The final command writes out an executable command file (/CMD). If we were to disassemble TEST/CMD, we would find that START is located at 0100H and BUFFER at 1000H. Thus the program is separated into ROM and RAM sections.

Relocatable assemblers and linkers have other capabilities, such as the use of COMMON blocks. You can also generate absolute code, if you use the ASEG pseudo-op.

Finally, we get to the actual format of a Microsoftrelocatable object file. A /REL file is composed of a bit (not byte) stream. Each /REL file may contain a table of ENTRY points and EXTERNAL references. Each ENTRY point is identified by its name (1 to 7 ASCII characters; although some releases of M80/L80 support only 1-6) and its relative location within one of the module's code, data, or common segments. Each EXTERNAL reference is identified by its name, and also by a chain (or linked list) of pointers, each of which locates the relative address within the module where the external was used. The last pointer in the chain is zero. The /REL file also contains internal relocation data necessary for resolution of label references within the module. All external and internal relocatable references are changed to absolute values at link time, when the program's segment origins have been established. The remainder of the information in the /REL file consists of absolue code and data bytes which do not need relocation, and numerous other fields which describe common blocks, the module name, the module segment lengths, and the /REL file end (or EOF byte). A library file would contain many such modules, each separated by program end indicators, and terminated by an EOF byte.

Let's take one Last look at our example, modified slightly, to see what the relocatable object file assembled from this source code would look like:

```
                          NAME   (`TEST')
     4470                 @DATE  EQU    4470H
     402D                 @EXIT  EQU    402DH
     4467                 @DSPLY        EQU    4467H
     000D                 CR     EQU    0DH
                                 CSEG   ;code starts here
                                 ENTRY  START
                                 EXT    MESSAGE
     0000' 21 0000"       START:        LD     HL,BUFFER
     0003' CD 4470                      CALL   @DATE
     0006' 3E 0D                 LD     A,CR
     0008' 32 0008"              LD     (BUFFER+8),A
     000B' 21 0000"              LD     HL,BUFFER
     000E' 11 0000*              LD     DE,MESSAGE
     0011' 01 0009                      LD     BC,BUFFLEN
     0014' ED BO                 LDIR
     0016' 21 0000*              LD     HL,MESSAGE
     0019' CD 4467                      CALL   @DSPLY
     001C' C3 402D                      JP     @EXIT
                                 DSEG   ;data starts here
                                 ENTRY  BUFFER
```

```
0000"                       BUFFER        DS     9
0009                        BUFFLEN EQU   $-BUFFER
                                    END   START

@DATE         4470  @DSPLY            4467  @EXIT402D
BUFFER              0000" BUFFLEN     0009  CR   000D
MESSAGE             0017* START       0000'
```

Notice how the external label, MESSAGE, is defined in the symbol table; the value 0017H represents the relative location of the LAST reference to MESSAGE in the assembled code, and the trailing asterisk (") denotes an external symbol both in this table and in the assembled code listing.

The following is a tabular picture of the decoded /REL file. Each column represents:

1. Absolute [0] or relocatable [1] item [1 bit]. If absolute, column (2) shows the value in hex [8 bits].

2. Relocation type [0 = special link item; 1, 2, or 3 = segment relative] [2 bits]. See column (8).

3. Special link item control field in decimal [4 bits]. See column (8).

4. "A-field" address type, same as column (2) [2 bits].

5. "A-field" value, displayed as high/low, but reversed in file [16 bits].

6. "B-field" length [3 bits].

7. "B-field" symbol in ASCII [8 bits each character].

Description of the object file record as decoded.

```
(1) (2)  (3)  (4)    (5) (6)   (7)                   (8)
--- ---  ---  ---  ------ --  -------  ---------------------------------
 1    0    2              4   TEST     program name
 1    0    0              5  START    entry symbol for library search
 1    0    0              6  BUFFER   entry symbol for library search
 1    0   10    0   00 09              define data area size
 1    0   13    1   00 1F              define program size
 1    0   11    1   00 00             set loading location counter (code)
 0   21                                absolute (1st byte in code segment)
 1    2             00 00             data relative (ref. to BUFFER)
 0   CD                                absolute
 0   70                         absolute
 0   44                         absolute
 0   3E                         absolute
 0   0D                         absolute
 0   32                         absolute
 1    2             00 08              data relative (ref. to BUFFER+8)
 0   21                         absolute
 1    2             00 00             data relative (ref. to BUFFER)
 0   11                                absolute (ref. to MESSAGE follows)
 0   00                                absolute (this plus next byte are
 0   00                                absolute  end of external chain)
 0   01                                absolute
```

```
 0   09                                     absolute
 0   00                                     absolute
 0   ED                                     absolute
 0   B0                                     absolute
 0   21                                     absolute (ref. to MESSAGE follows)
 0    1               00 0F                 program relative (link in chain)
 0   CD                                     absolute
 0   67                                     absolute
 0   44                                     absolute
 0   C3                                     absolute
 0   2D                                     absolute
 0   40                                     absolute
 1    0   11    2   00 00                  set loading location counter (data)
 1    0   11    2   00 09                   set loading location counter (data)
 1    0    7    2   00 00 6 BUFFER         define entry point (data)
 1    0    6    1   00 17 7 MESSAGE         chain external (head of list)
 1    0    7    1   00 00 5 START          define entry point (code)
 1    0   14    1   00 00                  end program (force to next byte
 1    0   15                                end file marker
```

What follows is a complete definition of the relocation format supported by this package.

The REL file is an encoded bit-stream containing relocatable object code information. It follows the format documented by Microsoft for the M8O assembler and L8O linker; however, only 16-bit externals are described.

1)IF the next bit is a zero, THEN the following eight bits are loaded according to the value of the location counter currently in effect, THEN recycle to 1).

ELSE IF the next bit is a one, THEN the next two bits represent a code which is interpreted as follows:

01 - Indicates a code relative value follows. The next 16 bits are loaded after being offset by the code segment origin, THEN recycle to 1).

10 - Indicates a data relative value follows. The next 16 bits are loaded after being offset by the data segment origin, THEN recycle to 1).

11 - Indicates a common relative value follows. The next 16 bits are loaded after being offset by the selected common segment origin, THEN recycle to 1).

00 - Indicates a Special Link item. The SL item consists of the following four bits which are interpreted as one of 16 different items described below; an optional VALUE field which consists of a 2-bit address type [00 = absolute, 01 = code relative, 10 = data relative, 11 = common relative] and a 16-bit address; and an optional NAME field that consists of a 3-bit name length followed by the name in 8-bit bytes. SLs 0000-0100 use only a NAME field; SLs 0101-1000 use both a VALUE field and a NAME field; SLs 1001-1110 use only a VALUE field; SL 1111 has neither a NAME nor a VALUE field. Unless otherwise specified, at the

conclusion of processing a special link item, processing recycles to 1). The Special Link items are as follows:

0000 - indicates an entry symbol. This is used by the linker only when searching a library to see if the module is needed to satisfy an undefined extern.

0001 - Select Common Block. Used to specify the NAMEd Common Block for subsequent common relative references.

0010 - Module name. This is the name of the module. The first one encountered is saved by MLINK for use in generating the optional HEADER record of the /CMD file.

0011 - Request Library Search. The library designated by the NAME field will be searched to resolve undefined externals prior to any object code generation. An REL will be first assumed. If one is not found, an IRL will then be assumed.

0100 - This item is reserved by Microsoft.

0101 - Define Common Size. This is used by MLZNK to establish the size of the common block designated by the NAME field.

0110 - Chain External. The VALUE field contains a pointer to the head of a chain which ends with an absolute zero. Each 16-bit element of the chain will be replaced with the value of the external symbol described in the NAME field.

0111 - Define Entry Point. The VALUE field specifies the value of the symbol described by the NAME field.

1000 - This item is reserved by Microsoft.

1001 - External plus Offset. This specifies that the VALUE field must be added to the following two bytes in the current segment after all chain externals have been processed.

1010 - Define Data Size. The VALUE field is used by the linker to establish the size of the data segment of the current module.

1011 - Set Location Counter. The location counter is set to the value identified by the VALUE field.

1100 - Chain Address. The VALUE field contains the the head pointer of a linked list; each entry in the list is to be replaced by the current value of the location counter. Chain address is generated by one-pass assemblers (or compilers) to have the linkerfixup forward references.

1101 - Define Code Size. The VALUE field is used by the linker to establish the size of the code segment of the current module.

1110 - End of Module. The VALUE field defines the transfer address for the module if other than absolute zero. This item denotes the end of the module. The bit stream is also advanced to a byte boundary. Recycle to 1) if loading a module from other than a library search.

1111 - End of File. This is used to indicate the end of the file. It is used when searching libraries or when loading modules to detect the end of the file.

Warranty

This software program(s) is warranted to perform as documented when used on the specified hardware operating under the specified disk operating system as shown on the accompanying documentation. If within 90 days of the date of purchase the program is found to be defective due to a bug in the code, the publisher will, upon request, provide a patch to correct the bug or will update the program diskette with a corrected copy within a reasonable time period after return of the program diskette to the publisher. If within 90 days of the date of purchase the documentation proves defective due to missing pages, the publisher will provide substitutes for the missing pages upon request.

The publisher shall have no liability or responsibility to the purchaser or any other person, company, or entity with respect to any liability, loss, or damage caused or alleged to have been caused by this product, including but not Limited to any interruption of service, loss of business and anticipatory profits, or consequential damages resulting from the operation or use of this program.

Customer Support

Customer service information on this product may be acquired by contacting MYSOSYS, Inc., at the following address:

MYSOSYS,Inc.
P.O. Box 239
Sterling, Virginia 22170-0239
703-450-4181