# Table of Contents

Table of Contents (Continued)

## 1.0 INTRODUCTION

RMA is a full feature relocatable macro assembler and linkage editor for OS-9. It was designed to be used by advanced programmers or in conjunctions with compiler systems.

RMA permits sections of assembly language programs to be independently assembled to "relocatable object files". The linkage editor takes any number of program sections and/or library sections and combines them into a single executable OS-9 memory module. Global data (including indexed and direct addressing modes) and program references are automatically resolved in the process. The macro facility permits commonly used statement sequences to be defined, then used within the program with appropriate parameter substitution. RMA also supports conditional assembly and library source files.

Some of the characteristics of RMA include:

1. Supports the OS-9's modular, multi-tasking environment.
2. Built-in functions for calling OS-9.
3. Supports use of position-independent, reentrant code.
4. Allows programs to be written and assembled separately and then linked together, which allows creation of standard subroutine libraries
5. Provides macro capabilities.
6. Compatible with either OS-9 Level I or Level II.

RMA is a two-pass assembler. During the first pass through the source, the symbol table is created by scanning each line and picking up the symbol definitions. During the second pass, machine language instructions and data values are placed in the relocatable object file. The linker combines previously assembled relocatable object files in a separate pass.

This manual describes how to use the OS-9 Relocating Macro Assembler and basic programming techniques for the OS-9 environment, but it is not intended to be a comprehensive course on 6809 assembly language programming. If you are not familiar with these topics, you should consult the Motorola 6809 programming manuals and one of the many assembly-language programming books available at bookstores and libraries.

## 1.1 THE ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT PROCESS

Writing and testing of assembly language programs using RMA involves a basic edit/assemble/link/test cycle. RMA can simplify this process because programs can be written in sections that can be assembled separately, then linked together to form the entire program. In the event one program section must be changed for any reason, only the revised section has to be reassembled.

Here is a summary of the steps involved in the RMA assembly language development process:

1. Create a source program file using the text editor.
2. Run the assembler to translate the source file(s) to a relocatable object module(s).
3. If the assembler reports errors, use the text editor to correct the offending source file, then go to step 2.
4. Run the linker to combine all required relocatable modules. If the linker reports errors, correct them and go to step 2.
5. Run and test the program. The OS-9 Interactive Debugger is frequently used to do this.
6. If the program has bugs, use the text editor to correct the source file, then go to step 2.
7. Document the program and you are done!

## 1.2 INSTALLATION

The RMA distribution disk contains a number of files that should be copied to the working system disk. The original distribution disk should then be stored in a safe place for backup purposes.

**RMA** is the assembler program, and **RLINK** is the linkage editor program. Both of these files should be copied to the system's execution ("CMDS") directory.

**Root.a** is an assembly language source code file which is a "front end" section for programs that use initialized data. It should be copied to an RMA working data directory.

## 1.3 CALLING AND RUNNING RMA

The Relocating Macro Assembler is a command program that can be run from the OS-9 Shell, from a Shell procedure file, or from another program. The disk file and memory module names are "RMA". The basic format of a command line to run the assembler is:

**RMA filename [option(s)] [ >listing ]**

Brackets enclose optional things, thus the only items absolutely required are the "RMA" command name, and "filename" which is the source text file name (or more correctly, pathlist). A typical command line looks like this:

**RMA prog5 -l -s -c >/p**

In this example the source program is read from the file "prog5". The source file name can be followed by an <u>option list</u>, which allows you to control various factors such as whether or not a listing or object file is to be generated, control the listing format, etc. The option list consists of one or more option abbreviations separated by spaces or commas. An option is turned on by its presence in the list preceded by a minus sign, or a double minus sign followed by an option abbreviation acts to turn the function off. If an option is not expressly given, the assembler will assume a <u>default</u> condition for it. Also, command line options can be overridden by OPT statements within the source program (see the OPT statement description for more information). In the example above, the options "l" and "s" are turned on, and "c" is turned off.

RMA handles memory allocation for its data area. Most of the data area memory is needed for the symbol data. RMA will take the memory that it needs up to the maximum available memory.

The final item, ">listing", allows the program listing generated by the assembler (on the standard output path) to be optionally redirected to another pathlist, which may be an output device such as a printer, a disk file, or a pipe to another program. Output redirection is handled by the Shell and not the assembler itself. If I/O redirection is omitted from the command line, the output will appear on your terminal display. In the above example, the listing output was directed to a device called "p", which is the name of the printer on most OS-9 systems.

## 1.4 RMA OPTIONS

Up to 10 options are allowed on the command line. Each option is specified by a single letter preceded by a "-" or "--". Use "-" to turn on an option and "--" to turn the option off. The recognized assembler options are:

**-o=<path>** Write relocatable output to the file specified (must be a mass storage file).

**-l** Write formatted assembler listing to standard output. If off, only error messages are printed. (Default off)

**-c** Suppress listing of conditional assembly lines in assembler listing. (Default on)

**-f** Form feed for top of form. Use form feed for page eject instead of line feeds. (Default off)

**-g** List all code bytes generated. (Default off)

**-x** Suppress macro expansion in listing. (Default on)

**-e** Suppress printing of errors. (Default on)

**-s** Print symbol table. Prints the entire contents of the symbol table at the end of the assembly listing. (Default off)

**-dn** Set listing lines per page to **n**. (Default 66)

**-wn** Set line width to **n**. Defines the maximum length of each line. Lines are truncated if they exceed this number. (Default 80)

## 1.5 INPUT FILE FORMAT

The OS-9 Assembler reads its input from an input file (path) which contains variable-length lines of ASCII characters. Input files may be created and edited by the OS-9 Macro Text Editor or any other standard text editor.

Each input line is a text string terminated by an end-of-line (return) character. The maximum length of the input line is 256 characters. Each line contains assembler statements as explained in this manual. The line can have from one to four "fields":

* an optional label field
* an operation field
* an operand field (for some operations)
* an optional comment field

There are also two special cases: if the first character of a line is an asterisk, the entire line is treated as a comment which is printed in the listing but not otherwise processed. Blank lines are ignored but are included in the listing.

### Label Field

The label field begins in the first character position of the line. Labels are usually optional (instructions), but there are exceptions. They are required by some statements (i.e. EQU and SET), or not allowed on others (assembler directives such as SPC, TTL, etc.). The first character of the line must be a space if the line does not contain a label. If a label is present, the assembler defines the label as the address of the first byte of where the object code of the instruction will be loaded. An exception to the rule is that labels on SET and EQU statements are given the value of the result of evaluation of the operand field.

If a symbolic name in the label field of a source statement is followed by a ":" (colon), the name will be known GLOBALLY by all modules that have been linked together. Since the label is known globally, a branch or jump can be done to a PSECT in another module. If no colon appears after the label, the label will be known only in the PSECT where it is defined. The PSECT statement is similar to MOD statement in the Microware Interactive Assembler.

The label must be a legal symbolic name consisting of from one to nine uppercase or lowercase characters, decimal digits, or the characters "$", "_", or ".", however the first character must be a letter (see Sect. 3.3). Upper and lower case characters are distinct.

Labels (and names in general) must be unique, i.e., they cannot be defined more than once in a program (except when used with the "SET" directive). An exception to this rule is that labels on SET and EQU statements are given the value of the result of evaluation of the operand field. In other words, these statements allow any value to be associated with a symbolic name. Labels on RMB statements are given the current value of the data address counter.

**The Operation Field**

This field specifies the machine language instruction or assembler directive statement mnemonic name. It immediately follows and is separated from the label field by one or more spaces.

Some instructions must include a register name which is part of the operation field (i.e., LDA, LDD, LDU). In these instructions the register name must be part of the name and <u>cannot</u> be separated by spaces. RMA accepts instruction mnemonic names in either uppercase or lowercase characters.

Instructions cause one to five bytes of object code to be generated depending on the specific instruction and addressing mode. Some assembler directive statements (such as FCB, FCC) also cause object code to be generated.

**Operand Field**

The operand field follows and must be separated by at least one space from the instruction field. Some instructions don't use an operand field; other instructions and assembler directives require one to specify an addressing mode, operand address, parameters, etc. The sections describing the instructions and assembler directives explain the format for operand(s), if any.

**Comment Field**

The last field of the source statement is the optional comment field which can be used to include a descriptive comment in the source statement. This field is not processed other than being copied to the program listing.

## 1.6 ASSEMBLY LISTING FORMAT

If the "-l" option is given in the RMA command line it, a formatted assembly listing will be written to the standard output path. The output listing the following format:

```
00098 0032 59         +         rolb
00117 0045=17ffb8       label   lbsr  _dmove      copy result
^     ^    ^^         ^ ^       ^     ^           ^
|     |    ||         | |       |     |           Comment area
|     |    ||         | |       |     Start of operand
|     |    ||         | |       Start of Mnemonic
|     |    ||         | Start of label
|     |    ||         A "+" indicates a macro expansion
|     |    |Start of object code bytes
|     |   An "=" here indicates operand has an external reference
|     Location counter value
Listing line sequence number
```

## 1.7 EVALUATION OF EXPRESSIONS

Operands of many instructions and assembler directives can include numeric expressions in one or more places. The assembler can evaluate expressions of almost any complexity using a form similar to the algebraic notation used in programming languages such as BASIC and FORTRAN.

Expressions consists of <u>operands</u>, which are symbolic names or constants, and <u>operators</u>, which specify an arithmetic or logical function. All assembler arithmetic uses two-byte (internally, 16 bit binary) signed or unsigned integers in the range of 0 to 65535 for unsigned numbers, or -32768 to +32767 for signed numbers.

In some cases, expressions are expected to evaluate to a value which must fit in one byte (such as 8-bit register instructions), and therefore must be in the range of 0 to 255 for unsigned values and -128 to 127 for signed values. In these cases, if the result of an expression is outside of this range an error message will be given.

Expressions are evaluated from left-to-right using the algebraic order of operations (i.e. multiplications and divisions are performed before additions and subtractions). Parentheses can be used to alter the natural order of evaluation.

### Expression Operands

The following items may be used as operands within an expression:

DECIMAL NUMBERS: optional minus sign and one to five digits, for example:

```
100
-32767
0
12
```

HEXADECIMAL NUMBERS: dollar sign ("$") followed by one to four hexadecimal characters (0-9, A-F or a-f), for example:

```
$EC00
$1000
$3
$0300
```

BINARY NUMBERS: percent sign ("%") followed by one to sixteen binary digits (0 or 1), for example:

```
%0101
%1111000011110000
%10101010
```

CHARACTER CONSTANTS: single quote ("´") followed by any printable ASCII character. For example:

```
´X
´c
´5
´c
```

SYMBOLIC NAMES: one to nine characters: upper and lower case alpha (A-Z, a-z), digits (0-9), and special characters _, ., @, or $ (underscore, period or dollar sign), the first character of which cannot be a digit.

INSTRUCTION COUNTER: the asterisk ("*") represents the program instruction counter value as of the beginning of the line.

## Expression Operators

Operators used in expressions operate on one operand (negative and not) or on two operands (all others). The table below shows the available operators, listed in the order they are evaluated relative to each other, e.g, logical OR operations are performed before multiplications. Operators listed on the same line have identical precedence and are processed from left to right when they occur in the same expression.

Assembler Operators By Order of Evaluation

| | |
|---|---|
| - negative | ^ logical NOT |
| & logical AND | ! logical OR |
| * multiplication | / division |
| + addition | - subtraction |

Logical operations are performed bitwise, i.e., the logical function is performed bit-by-bit on each bit of the operands. Division and multiplication functions assume <u>unsigned</u> operands, but subtraction and addition work on signed (2´s complement) or unsigned numbers. Division by zero or multiplication resulting in a product larger than 65536 have undefined results and are reported as errors.

## Symbolic Names

A symbolic name consists of from one to nine uppercase or lowercase characters, decimal digits, or the characters "$", "_", or ".", and "@". However, the first character must be a letter. The following are examples of legal symbol names:

```
HERE        there         SPL030       PGM_A
Q1020.1     t$integer     L.123.X      a002@
```

One thing to keep in mind is that the Relocating Macro Assembler does not fold lowercase letters to uppercase. The names "file_A" and "FILE_A" are distinct names.

These are examples of some illegal symbol names with reasons why they are such:

```
2move          - does not start with a letter
main.backup    - more than 9 characters
lbl#123        - # is not a legal name character
```

Names are defined when first used as a label on an instruction or directive statement. They must be defined exactly one time in the program (except SET labels: see SET statement description). If a name is redefined (used as a label more than once) an error message is printed on subsequent definition(s).

If a symbolic name is used in an expression and has not been defined, the Relocating Macro Assembler assumes the name is external to the PSECT. Information will be recorded about the reference so the linker can adjust the operand accordingly. However, external names cannot appear in operand expressions for assembler directives.

This Page Intentionally Left Blank

## 2.0 INTRODUCTION TO MACROS

Sometimes identical or similar sequences of instructions may be repeated in different places in a program. The problem is that if the sequence of instructions is long or must be used a number of times, writing it repeatedly can be tedious.

A macro is a definition of an instruction sequence that can be used numerous places within a program. The macro is given a name which is used similarly to any other instruction mnemonic. Whenever RMA encounters the name of a macro in the instruction field, it outputs all the instructions given in the macro definition. In effect, macros allow the programmer to create "new" machine language instructions.

For example, suppose a program frequently must perform 16 bit left shifts of the D register. This two-instruction sequence can be defined as a macro, for example:

```
dasl    macro
        aslb
        rola
        endm
```

The "macro" and "endm" directives specify the beginning and the end of the macro definition, respectively. The label of the "macro" directive specifies the name of the macro, "dasl" in this example. Now the "new" instruction can be used in the program:

```
        ldd 12,s        get operand
        dasl            double it
        std 12,s        save operand
```

In the example above, when RMA encountered the "dasl" macro, it actually outputted code for "aslb" and "rola". Normally, only the macro name is listed as above but an RMA option can be used to cause all instructions of the "macro expansion" to be listed.

Macros should not be confused with subroutines although they are similar in some ways. Macros repetitively duplicate an "in line" code sequence every time they are used and allow some alteration of the instruction operands. Subroutines appear exactly once, never change, and are called using special instructions (BSR. JSR, and RTS). In those cases where they can be used interchangeably, macros usually produce longer but slightly faster programs, and subroutine produce shorter and slightly slower programs. Short macros (up to 6 bytes or so) will almost always be faster and shorter than subroutines because of the overhead of the BSR and RTS instructions needed.

## 2.1 MACRO STRUCTURE

A macro definition consists of three sections:

1. The macro header - assigns a name to the macro
2. The body - contains the macro statements
3. The terminator - indicates the end of the macro

```
<name>  MACRO    /* macro header */
        .
        .
        body     /* macro body */
        .
        .
        ENDM     /* macro terminator */
```

The macro name must be defined by the label given in the **MACRO** statement. The name can be any legal assembler label. It is possible to <u>redefine</u> the 6809 instructions (LDA, CLR, etc.) themselves by defining macros having identical names. This gives RMA the capability to be used as a "cross-assembler" for non-6809 8 or 16 bit processors by definition and/or redefinition of the instruction set of the target CPU. Caution: redefinition of assembler directives such as "RMB" can have unpredictable consequences.

The body of the macro can contain any number of legal RMA instruction or directive statements including references to previously-defined macros. The last statement of a macro definition must be **ENDM**.

The text of macro definitions are stored on a temporary file created and maintained by RMA. This file has a large (1K byte) buffer to minimize disk accesses. Therefore, programs that use more than 1K of macro storage space should be arranged so that short, frequently used macros are defined first so they are kept in the memory buffer instead of disk space.

Macro calls may be nested, that is, the body of a macro definition may contain a call to another macro. For example:

```
times4 MACRO
       dasl
       dasl
       ENDM
```

The macro above consists of the "dasl" macro used twice. The definition of a new macro within another is not permitted. Macro calls may be nested up to eight deep.

## 2.2 MACRO ARGUMENTS

Arguments permit variations in the expansion of a macro. Arguments can be used to specify operands, register names, constants, variables, etc., in each occurence of a macro.

A macro can have up to nine formal arguments in the operand fields. Each argument consists of a backslash character and the sequence number of the formal argument, e.g, \1, \2 ... \9. When the macro is expanded, each formal argument is replaced by the corresponding text string "actual argument" given in the macro call. Arguments can be used in any part of the operand field not in the instruction or label fields. Formal arguments can be used in any order and any number of times.

For example, the macro below performs the typical instruction sequence to create an OS-9 file:

```
create  MACRO
        leax \1,pcr     get addr of file name string
        lda  #\2        set path number
        ldb  #\3        set file access modes
        os9 I$CREATE
        ENDM
```

This macro uses three arguments: "\1" for the file name string address; "\2" for the path number; and "\3" for the file access mode code. When "create" is referenced, each argument is replaced by the corresponding string given in the macro call, for example:

```
        create outname,2,$1E
```

The macro call above will be expanded to the code sequence:

```
        leax outname,pcr
        lda  #22
        ldb  #$1E
        os9  I$CREATE
```

If an argument string includes special characters such as backslashes or commas, the string must be enclosed in double quotes. For example, this macro reference has two arguments:

```
        double  count,"2,s"
```

An argument may be declared null by omitting all or some arguments in the macro call. This makes the corresponding argument an empty string so no substitution occurs when it is referenced.

There are two special argument operators that can be useful in constructing more complex macros. They are:

\Ln - Returns the length of the actual argument n, in bytes.

\# - Returns the number of actual arguments passed in a given macro call.

These special operators are most commonly used in conjunction with RMA's conditional assembly facilities to test the validity of arguments used in a macro call, or to change the way a macro works according to the actual arguments used. When macros are performing error checking they can report errors using the FAIL directive. Here is an example using the "create" macro given on the previous page but expanded for error checking:

```
create  MACRO
        ifne \# - 3      must have exactly 3 args
        FAIL create: must have three arguments
        endc
        ifgt \L1 - 29   file name can be 1 - 29 chars
        FAIL create: file name too long
        endc
        leax \1,pcr     get addr of file name string
        lda  #\2        set path number
        ldb  #\3        set file access modes
        os9 I$CREATE
        ENDM
```

## 2.3 MACRO AUTOMATIC INTERNAL LABELS

Sometimes it is necessary to use labels within a macro. Labels are specified by "\@". Each time the macro is called, a unique label will be generated to avoid multiple definition errors. Within the expanded code "\@" will take on the form "@xxx", where xxx will be a decimal number between 000 to 999.

More than one label may be specified in a macro by the addition of an extra character(s). For example, if two different labels are required in a macro, they can be specified by "\@A" and "\@B". In the first expansion of the macro, the labels would be "@001A" and "@001B", and in the second expansion they would be "@002A" and "002B". The extra characters may be appended before the "\" or after the "@".

Here is an example of macro that uses internal labels:

```
testovr MACRO
        cmpd    #\1     compare to arg
        bls     \@A     bra if in range
        orcc    #1      set carry bit
        bra     \@B     and skip next instr.
\@A     andcc   #$FE    clear carry
\@B     equ     *       continue...
```

Suppose the first macro call is:

```
        testovr $80
```

The expansion will be:

```
        cmpd    #$80    compare to arg
        bls     @001A   bra if in range
        orcc    #1      set carry bit
        bra     @001B   and skip next instr.
@001A   andcc   #$FE    clear carry
@001B   equ     *       continue...
```

If the second macro call is:

```
        testovr 240
```

The expansion will be:

```
        cmpd    #240    compare to arg
        bls     @002A   bra if in range
        orcc    #1      set carry bit
        bra     @002B   and skip next instr.
@002A   andcc   #$FE    clear carry
@002B   equ     *       continue...
```

## 2.4 ADDITIONAL COMMENTS ABOUT MACROS

Macros can be an important and useful programming tool that can significantly extend RMA's capabilities. In addition to creating instruction sequences, they can also be used to create complex constant tables and data structures.

Macros can also be dangerous in the sense that if they are used indiscriminately and unnecessarily they can impair the readability of a program and make it difficult for programmers other than the original author to understand the program logic. Therefore, when macros are used they should be carefully documented.

## 3.0 PROGRAM SECTIONS

A primary purpose of RMA is to permit programs to be composed of different segments that can be assembled separately. The linker (RLINK) combines all of the segments into a single OS-9 memory module with a coordinated data storage area. By use of external names, code in each segment can reference variables declared in other segment, or may transfer program control (using BRA, BSR, etc.) to labels in other segments.

When the assembler source program for each segment is written, it must be divided into distinct **program sections** for variable storage definitions (VSECTs) and for program statements (PSECTs). The output of the assembler is a distinct **relocatable object file** (ROF) which contains the object code output plus information about the variable storage declarations for use by the linker.

The linker reads the ROFs and assigns space in the data storage area and combines all the object code into a single executable memory module. As it does so, it must alter the operands of instructions to refer to the final variable assignments and must also adjust program transfer control instructions that refer to label in other segments.

For example, if three segments called "A", "B", and "C", respectively, are processed by the linker, a vastly simplified memory map of the product would look like this:

```
            Process Data Area

      +------------------------+
      |  Segment A Variables   |  These correspond to each
      +------------------------+  segment's VSECTs
      |  Segment B Variables   |
      +------------------------+
      |  Segment C Variables   |
      +------------------------+

         Executable Memory Module

      +------------------------+
      |     Module Header      |  <- Generated by RLINK
      +------------------------+
      | Segment A Object Code  |  <- "Mainline" segment
      +------------------------+
      | Segment B Object Code  |  These correspond to each
      +------------------------+  segment's PSECTs
      | Segment C Object Code  |
      +------------------------+
      |    CRC Check Value     |  <- Generated by RLINK
      +------------------------+
```

## 3.1 PROGRAM SECTION DECLARATIONS: PSECT, VSECT, CSECT

RMA uses three section directives (PSECT, VSECT, and CSECT) to control the placement of object code and allocation of variable space in the program. The end of each section is indicated by the ENDSECT directive.

PSECT indicates to the linker the beginning of a relocatable-object-format (ROF) file, initializes the instruction and data location counters, and assembles subsequent instructions into the ROF object code area.

VSECT causes the Relocating Macro Assembler to change to the variable (data) location counters and places information about subsequently declared variables into the appropriate ROF data description area. VSECTs are declared within PSECTs.

CSECT initializes a base value for assigning sequential numeric values to symbolic names. CSECTS are provided as a convenience to the programmer and their use is entirely optional.

Each section contains the following location counters:

```
PSECT - instruction location counter

VSECT - initialized direct page counter
        non-initialized direct page counter
        initialized data location counter
        non-initialized data location counter

CSECT - base offset counter
```

The source statements placed in a particular section cause the linker to perform a function appropriate for the statement. Therefore, the type of mnemonics allowed within a section are sometimes restricted except for the following instructions which may appear inside or outside any section: nam, opt, ttl, pag, spc, use, fail, rept, endr, ifeq, ifne, iflt, ifle, ifge, ifgt, ifpl, endc, else, equ, set, macro, endm, and endsect.

### 3.1.1 PSECT DIRECTIVE

Syntax: **PSECT** name,typelang,attrev,edition,stacksize,entrypoint

Legal PSECT Statements: any 6809 instruction mnemonic, fcc, fdb, fcs, fcb, rzb, vsect, endsect, os9, end. Warning: RMB cannot be used within a PSECT

PSECT is the program code section. There can only be one PSECT per assembly language file. The PSECT directive initializes all assembler location counters and marks the start of the program segment. All instruction statements and VSECT data reservations (RMBs) must be declared within the PSECT/ENDSECT block.

PSECT may have an operand list containing a name and five expressions if the PSECT is to be a "mainline" segment, or it can have no operand list at all. If an operand list is provided, the operand list will be stored in the ROF for later use by the linker to generate the memory module header. If no operand list is provided, the PSECT name defaults to "program" and all other expressions to zero. The elements of the PSECT operand list are as follows:

name — Up to 20 bytes for a name to be used by the linker to identify the PSECT. Any printable character may be used except a space or a comma. The name does not need to be unique from other PSECT names, but it is easier to identify PSECTs that the linker has problems with if the names are different.

typelang — A byte expression to be used as the executable module type/language byte. If the PSECT is not a mainline segment the type/language byte must be zero.

attrrev — A byte expression to be used as the executable module attribute/revision byte.

edition — A byte expression to be used as the executable module attribute/revision byte.

stacksize — A word expression that estimates the amount of stack storage required by this PSECT. The linker totals the value in all PSECTs to appear in the executable module and adds the value to any data storage requirement for the entire program.

entry — A word expression to be used as the program entry point offset for PSECT goes here. If the PSECT is not a mainline segment, this should be 0.

An important difference between PSECT and the MOD statement in the Microware Interactive Assembler is that MOD directly outputs an OS-9 module header, while PSECT sets up information for the linker which create the module header when it is run later.

Example of the use of a PSECT

```
* this program starts a basic09 process

          ifp1
          use ..../defs/os9defs.a
          endc

PRGRM     equ $10
OBJCT     equ $1

stk       equ 200
          psect rmatest,$11,$81,0,stk,entry

name      fcs /basic09/
prm       fcb $d
prmsize   *-prm

entry     leax name,pcr
          leau prm,pcr
          ldy  #prmsize
          lda  #PRGRM+OBJCT
          clrb
          os9 F$FORK
          os9 F$WAIT
          os9 F$EXIT
          endsect
```

PSECT

vsect

ds.b

ends

ends

### 3.1.2 VSECT DIRECTIVE

Syntax: **VSECT** [DP]

ds.b - .w .l

Legal Internal Statements: rmb, fcc, fdb, fcs, fcb, rzb, endsect.

VSECT is the variable storage section which can contain either initialized or uninitialized variable storage definitions. VSECT directive causes RMA to change to the data location counters. There are two sets of counters, one set for direct page variables, and another set for other variables which are normally index-register offsets into a process's data storage area. If "DP" appears after VSECT, the direct page counters are used, otherwise the index register counters are used.

The RMB directive within the VSECT section reserves the specified number of bytes in the uninitialized data area. RMB can only be used within a VSECT.

The fcc, fdb, fcb, fcs, and rzb (reserve zeroed bytes) directives place data into the initialized data area. Initialized constants that appear inside a VSECT must be moved from the data section to the program section so they can be accessed using the 6809 program-counter relative addressing mode. Initialized constants can appear outside of a VSECT but if they do, they cannot be changed in any way. Any number of VSECT blocks can be in a PSECT. Note, however, that the data location counters maintain their values from each VSECT block to the next. Since the linker handles the actual data allocation, there is no facility to adjust the data location counters.

An example of VSECT usage is given on the following page.

## VSECT Example Program

```
        ifpl
        use ..../defs/os9defs.a
        endc

PRGRM   EQU $10
OBJCT   EQU $1
stk     EQU 200
        PSECT pgmlen,$11,$81,0,stk,start

       * data storage declarations
        VSECT
 temp   RMB 1
 addr   RMB 2
 buffer RMB 500
        ENDSECT

 start  leax buffer,u get address of buffer
        clr temp
        inc temp
        ldd #500 loop count
 loop   clr ,x+
        subd #1
        bne loop
        os9 F$EXIT  return to OS9
        ENDSECT
```

### 3.1.3 CSECT DIRECTIVE

Syntax: CSECT {expression}

The CSECT directive provides a means for assigning consecutive offsets to labels without resorting to EQUs. If an expression is present, the CSECT base counter is set to the given value. Otherwise, the base counter is set to zero.

CSECT Example

```
          CSECT 0
R$CC      rmb   1  Condition code register
R$A       rmb   1  A Accumulator
R$B       rmb   1  B Accumulator
          ENDSECT
```

The above CSECT will assign offsets of 1, 2, and 3 respectively.
See the Defs file for CSECT examples.

This Page Intentionally Left Bank

## 4.1 ASSEMBLER DIRECTIVE STATEMENTS

Assembler directive statements give the assembler information that affects the assembly process, but do not cause code to be generated. Read the descriptions carefully because some directives require labels, labels are optional on others, and a few can not have labels.

## 4.2 END STATEMENT

Syntax: **END**

Indicates the end of a program. Its use is optional since END will be assumed upon an end-of-file condition on the source file. END statements may not have labels.

## 4.3 EQU and SET STATEMENTS

Syntax: label **EQU** <expression>
        label **SET** <expression>

These statements are used to assign a value to a symbolic name (the label) and thus require labels. The value assigned to the symbol is the value of the operand, which may be an expression, a name, or a constant. They can be used in any program section.

The difference between the EQU and SET statements is that:

- Symbols defined by EQU statements can be defined only once in the program

- Symbols defined by SET statements can be redefined again by subsequent SET statements.

In EQU statements the label name must not have been used previously, and the operand cannot include a name that has not yet been defined (i.e., it cannot contain as-yet undefined names whose definitions also use undefined names). Good programming practice, however, dictates that all equates should be at the beginning of the program.

EQU is normally used to define program symbolic constants, especially those used in conjunction with instructions. SET is usually used for symbols used to control the assembler operations, especially conditional assembly and listing control.

Examples:

```
FIVE    equ     5
OFFSET  equ     address-base
TRUE    equ     $FF
FALSE   equ     0
SUBSET  set     TRUE

        ifne    SUBSET
        use     subset.defs
        else
        use     full.defs
        endc
SUBSET  set     FALSE
```

## 4.4 FAIL STATEMENT

Syntax: **FAIL** textstring

This statement forces an assembler error to be reported. The "textstring" operand is displayed as the error message which is processed in the same manner as RMA-generated error messages. Because the entire line following the FAIL keyword is assumed to be the error message, this statement cannot have a comment field.

FAIL is most commonly used in conjunction with conditional assembly directives that the programmer sets up to test for various illegal conditions, especially within macro definitions.

Example:

```
IFEQ   maxval
FAIL   maxval cannot be zero
ENDC
```

## 4.5 IF, ELSE, and ENDC STATEMENTS

```
Syntax:  IFxx  <expression>
            <statements>
         [ELSE]
            <statements>
         ENDC
```

An important feature of the Relocating Macro Assembler is its conditional assembly capability. Simply stated, this is the ability to selectively assemble or not assemble one or more parts of a program depending on a variable or computed value. Thus, a single source file can be used to selectively generate multiple versions of a program.

Conditional compilation uses statements similar to the branching statements found in high level languages such as Pascal and Basic. The generic IF statement is the basis of this capability. It has as an operand a symbolic name or an expression. A comparison is made with the result: if the result of the comparison is true, statements following the IF statement will be processed. If the result of the comparison is false, the following statements will not be processed until an ENDC (or ELSE) statement is encountered. Hence, the ENDC statement is used to mark the end of a conditionally assembled program section. Here is an example that uses the IFEQ statement which tests for equality of its operand with zero:

```
IFEQ SWITCH
ldd #0          assembled only if SWITCH = 0
leax 1,x
ENDC
```

The ELSE statement allows the IF statement to explicitly select one of two program sections to assemble depending on the truth of the IF statement. Statements following the ELSE statement are processed only if the result of the comparison was false. For example:

```
IFEQ SWITCH
ldd #0       assembled only if SWITCH = 0
leax 1,x
ELSE
ldd #1       assembled only if SWITCH is not = 0
leax -1,x
ENDC
```

(continued)

**IF/ELSE/ENDC STATEMENTS** - Continued

Multiple IF statements may be used, and "nested" within other IF statements if desired. They cannot, however, have labels.

There are several kinds of IF statements, each performing a different comparison. They are:

| | |
|---|---|
| IFEQ | True if operand equals zero |
| IFNE | True if operand does not equal zero |
| IFLT | True if operand is less than zero |
| IFLE | True if operand is less than or equal to zero |
| IFGT | True if operand is greater than zero |
| IFGE | True if operand is greater than or equal to zero |
| IFP1 | True only during first assembler pass (no operand) |

The IF statements that test for less than or greater than can be used to test the relative value of two symbols if they are subtracted in the operand expression, for example,

```
IFLE    MAX-MIN
```

will be true if MIN is <u>greater</u> then MAX. Note the reversal of logic due to the fact that this statement literally means.

```
IF MAX-MIN <= 0
```

The IFP1 statement causes subsequent statements to be processed during pass 1, but skipped during pass 2. It is useful because it allows program sections which contain only symbolic definitions to be processed only once during the assembly. The first pass is the only pass during which they are actually processed because they do not generate actual object code output. The OS9Defs file is an example of a rather large section of such definitions. For example, the following statement is used at the beginning of many source files.

```
IFP1
use     /d0/defs/OS9Defs
ENDC
```

## 4.6 NAM and TTL STATEMENTS

Syntax: **NAM** string
**TTL** string

These statements allow the user to define or redefine a program name and listing title line which will be printed on the first line of each listing page's header. These statements CANNOT have label or comment fields.

The program name is printed on the left side of the second line of each listing page, followed by a dash, then by the title line. The name and title may be changed as often as desired.

Examples:

```
nam Datac
ttl Data Acquisition System
```

Generates:

```
Datac - Data Acquisition System
```

## 4.7 OPT STATEMENT

Syntax: **OPT** <option>

Allows any of several assembler control options to be set or reset. The operand of the OPT statement is one of the characters that represent the various options. An option is denoted by a single character with no "-" or "--". Two exceptions are the "D" and "W" options which must be followed by a number "num". This statement must not have label or comment fields. See page 1-5 for a description of the options available.

**Examples:**

```
opt l
opt w72
opt s
```

## 4.8 PAG and SPC STATEMENTS

Syntax: **PAG[E]**
**SPC** <expression>

These statements are used to improve the readability of program listings. They are not themselves printed, and cannot have labels.

The PAG statement causes the assembler to begin a new page of the listing. The alternate form of PAG is PAGE for Motorola compatibility.

The SPC directive puts blank lines in the listing. The number of blank lines to be generated is determined by the value of the operand, which can be an expression, constant, name. If no operand is used a single blank line is generated.

## 4.9 REPT AND ENDR STATEMENTS

Syntax: **REPT** <expr>
<statements>
**ENDR**

This statement can repeat the assembly of a sequence of instructions a specified number of times. The result of the operand expression is used as the repeat count. The expression cannot include EXTERNAL or undefined symbols. REPT loops cannot be nested.

Examples:

```
* Make module size exactly 2048 bytes
    REPT 2048-*-3   compute fill size w/crc space
    fcb 0
    ENDR
    emod

* 20 cycle delay
    REPT 5
    nop
    nop
    ENDR
```

## 4.10 RMB STATEMENT

Syntax: [label] RMB expr

RMB has two primary uses. First, it is used within VSECTs to declare storage for uninitialized variables in the data area. Second, it can be used in a CSECT to assign a sequential value to the symbolic name given as its label.

When RMB is used to declare variables, a label is usually given which is assigned the relative address of the variable. In OS-9, the address must not be absolute so indexed or direct page addressing modes are usually used to access variables. The actual relative address is not actually assigned until the linker processes the ROF. The expression given specifies the size of the variable in bytes. This value is added to the address counters in order to update them.

When RMBs are used in CSECTs, the label specified is given the current CSECT location counter value, then the counter is updated by adding the result of the expression given.

## 4.11 USE STATEMENT

Syntax: **USE** pathlist

Causes the assembler to temporarily stop reading the current input file. It then requests OS-9 to open another file/device specified by the pathlist, from which input lines are read until an end-of-file occurs. At that point, the latest file is closed, and the assembler resumes reading the previous file from the statement following the USE statement.

USE statements can be nested (e.g., a file being read due to a USE statement can also perform USE statements) up to the number of simultaneously open files the operating system will allow (usually 13, not including the standard I/O paths). Some useful applications of the USE statement are to accept interactive input from the keyboard during assembly of a disk file (as in USE /TERM); and including library definitions or subroutines into other programs. USE statements cannot have labels.

This Page Intentionally Left Bank

## 5.0 PSEUDO-INSTRUCTIONS

"Pseudo-instructions" are special assembler statements that generate object code but do not correspond to actual 6809 machine instructions. Their primary purpose is to create special sequences of constant data to be included in the program. Labels are optional on pseudo-instructions.

## 5.1 FCB and FDB STATEMENTS

Syntax: **FCB** <expression> {, <expression>}
**FDB** <expression> {, <expression>}

The Form Constant Byte and Form Double Byte pseudo-instructions generate sequences of single (FCB) and double (FDB) constants within the program. The operand is a list of one or more expressions which are evaluated as constants. If more than one constant is to be generated, the expressions are separated by commas.

FCB will report an error if an expression has a value of more than 255 or less that -128 (the largest number representable by a byte). If FDB evaluates an expression with an absolute value of less than 256 the high order-byte will be zero.

Examples:

```
FCB 1,20,'A

fcb index/2+1,0,0,1

FDB 1,10,100,1000,10000

fdb $F900,$FA00,$FB00,$FC00
```

If the FCB or FDB appears in a VSECT, the data is assigned to the appropriate initialized data area (DP or non-DP). Otherwise, the constant is placed in the code area. If the constant contains an external reference, the program (using "Root.a") must copy out and adjust the references.

string) dc.b
dc.b

## 5.2 FCC and FCS STATEMENTS

Syntax: **FCC** string dc.b
**FCS** string

These pseudo-instructions generate a series of bytes corresponding to a string of one or more characters operand. The output bytes are the literal numeric value of each ASCII character in the string. FCS is the same as FCC except the most significant bit (the sign bit) of the last character in the string is set, which is a common OS-9 programming technique to indicate the end of a text string without using additional storage.

If the FCB or FDB appears in a VSECT, the data is assigned to the appropriate initialized data area (DP or non-DP). Otherwise, the constant is placed in the code area. If initialized data is to be modified by the program, the declaration of the data must appear in a VSECT. It is necessary to set up the initialized data and adjust any addresses in the initialized data to reflect the absolute addresses of the reference.

The character string must be enclosed by delimiters before the first character and after the last character. The characters that can be used as delimiters are:

```
! " # $ % & ´ ( ) * + , - . /
```

Both delimiters must be the same character and cannot be included in the string itself. Examples:

```
FCC /most programmers are strange people/

FCS ,0123456789,

FCS AA     null string

FCC $z$

FCS ""     null string
```

If the FCS or FCC appears in a VSECT, the data is assigned to the appropriate data area (DP or non-DP). Otherwise, the constant is placed in the code area.

## 5.3 RZB STATEMENT - RESERVE ZERO BYTES

SYNTAX: **RZB** <expression>

This statement is used to fill memory with a sequence of bytes filled each having a value of zero. The 16 bit expression is evaluated and that number of zero bytes are placed in the appropriate code or data section.

## 5.4 OS9 STATEMENT

Syntax: **OS9** <expression>

This statement is a convenient way to generate OS-9 system calls. It has an operand which is a byte value to be used as the request code. The output is equivalent to the instruction sequence:

```
SWI2
FCB operand
```

A file called "OS9Defs", which is distributed with each copy of OS-9, contains standard definitions of the symbolic names of all the OS-9 service requests. These names are commonly used in conjunction with the OS9 statement to improve the readability, portability, and maintainability of assembly language software.

Examples:

```
OS9 I$READ          (call OS-9 "READ" service request)

os9 F$EXIT          (call OS-9 "EXIT" service request)
```

This Page Intentionally Left Blank

## 6.0 ACCESSING THE DATA AREA

In general, RMA assumes that the program data area will be accessed using indexed or direct page addressing modes. By convention, one index register will contain the starting address of the data area and the direct page register will contain the page number of the lowest-address page of the data area. The RMA/RLINK system is designed to automatically adjust operands of instructions using indexed and direct page addressing modes.

The data area will be accessed differently depending on if your program uses initialized data or not. Initialized data is data that has an initial value that will be modified by the program, and is created by FCB, FDB, FCC, and similar directive used in a VSECT. If no initialized data is used, the program data area will be accessed using index registers in the same manner as is done by programs assembled with the Microware Interactive Assembler.

## 6.1 PROGRAMS WITH NO INITIALIZED DATA

Programs that do not use initialized data declare all data storage in VSECTs using RMBs only. The diagram shown below shows how the data memory area and registers will be set up for a new process.

```
+-------------------+   <-- Y  (Highest address)
|                   |
|     parameter     |
|       area        |
|                   |
+-------------------+   <-- X, SP
|                   |
|     data area     |
|                   |
+-------------------+
|                   |
|    direct page    |
|                   |
+-------------------+   <-- U, DP  (Lowest address)
```

When the process is executed by OS-9, the MPU registers will contain the bounds of the data area: U will contain the beginning address, and Y will contain the ending address. The SP register is set to the ending address+1, unless parameters are used. The direct page register will be set to the page number of the beginning page. If there are no parameters, Y, X, and SP will be the same. Shell will always pass at least an end of line character in the parameter area.

1. If the U register is maintained throughout the program, constant-offset-indexed addressing can be used.

2. Part of the program's initialization routine can compute the actual addresses of the data structures and store these addresses in pointer locations in the direct page. The addresses can be obtained later using direct-page addressing mode instructions.

Important note: You cannot use program-counter relative addressing to obtain addresses of objects in the data section due to the fact that the memory addresses assigned to the program section and the address section are not a fixed distance apart.

## 6.2 USING INITIALIZED DATA

If you plan on using initialized data it will be necessary to copy the data from the initialized data section in the object module to the data storage area pointed at by the U register by using the "Root.a" mainline module (object code that is directly executable by an OS-9 F$FORK). Its function is to use the initializing values and offsets of initialized data location stored in the object code module to actually initialize variables. The initialization information area of the object code module is automatically generated by the linker based in information passed by the assembler in ROFs. A copy of "Root.a" is included in the RMA software package.

"Root.a" will set the Y register to point to the same location as U used to point to (the bottom of the data area). X will point to the parameter area, and U will point to the top of data allocated by the linker. The data-index register choice is arbitrary but must be used consistently. To maintain compatibility with code produced by the C compiler, the Y register is used as the data pointer. For more information about "Root.a", study the extensively commented source code supplied with RMA. The diagram shown on the next page explains how the data area is set up.

Process Data Area Layout

```
+--------------------+
|     parameters     |
+--------------------+  <--  X
|                    |
|       stack        |  <--  SP register
|                    |
+--------------------+
|    free memory     |
+--------------------+
|  requested memory  |
+--------------------+  <--  U
|   uninitialized    |
|        data        |
+--------------------+
|    initialized     |
|        data        |
+--------------------+
|   uninitialized    |
|    direct page     |
+--------------------+
|    initialized     |
|    direct page     |
+--------------------+  <--  Y,DP
```

Process Object Code Module Layout

```
+--------------------+
|  CRC Check Bytes   |
+--------------------+  ---
| Initializing Data  |   |
|      Offsets       |   |
+--------------------+  Used by Root.a
| Initializing Data  |   |
|       Values       |   |
+--------------------+  ---
| Additional (user)  |
| Executable PSECTS  |  <-   PC (this area)
+--------------------+
|  Mainline PSECT_   |
|      (Root.a)      |  <-   Program entry
+--------------------+        point
|   Standard OS-9    |
|   Module Header    |
+--------------------+
```

## 7.0 INTRODUCTION

The Relocating Macro Assembler (RMA) allows programs to be written, assembled separately, and then linked together to form a single object code OS-9 module. The function of the linker (RLINK) is to combine different relocatable object files (ROFs) into a single OS-9 format memory module and resolve external data and program references. The linker allows references to occur between modules so one module can reference a symbol in another module. This involves adjustment of the operand parts of many machine-language instructions.

During the assembly process when an external reference is made from one program section to another program section, the first program section has no idea where the external symbol it is referencing will be in the final memory module. When an external reference is encountered, the assembler will set up information denoting the external reference.

The linker takes as its input a ROF produced by the assembler. When the linker gets the file, no absolute addresses have been resolved. Relocatable means that no absolute addresses have been assigned. Each section is assembled as though it had started at absolute address 0. The linker will read in all of the ROF files and assign each one an absolute memory address for data locations and instruction locations for branching. All other absolute addresses are resolved by OS-9 at execution time.

The Relocating Macro Assembler and Linker allow the programmer to break a program up so that it can be written and debugged easier. When an error occurs in a program, only the module that has the error must be edited, reassembled, and relinked with the rest of the program.

## 7.1 RUNNING THE LINKER

The linker turns Relocating Macro Assembler output into executable form. All input files must be in relocatable object format (ROF). The linker is called using the command line:

link16

**rlink** [options] <mainline> [<subl> {<subn>} ] [options]

<Mainline> specifies the pathlist of the mainline segment from which external references are resolved and a module header is generated. A mainline module is indicated by setting the type/lang value in the PSECT directive to a non-zero value. Names of additional ROFs to be used in the linkage process (<subl>..<subn>) follow the mainline ROF pathlist. No other ROF can contain a mainline PSECT. The mainline and all subroutine files will appear in the final linked object module whether actually referenced or not.

It is the mainline module's job to perform the initialization of data and the relocation of any initialized data references within the initialized data using the information in the object module supplied by rlink (see chapter 6).

## 7.2 LINKER COMMAND LINE OPTIONS

The following options can appear on the command line:

| | |
|---|---|
| -o=<path> | Write linker object (memory module) output to the file specified. The last element in <path> is used as the module name unless overridden by -n. |
| -n=<name> | Use <name> as object file name. |
| -l=<path> | Use <path> as library file. A library file consists of one or more merged assembly ROF files. Each psect in the file is checked to see if it resolves any unresolved references. If so, the module is included in the final output module, otherwise it is skipped. No mainline psects are allowed in a library file. Library files are searched in the order given on the command line. |
| -E=<n> or -e=<n> | <n> is used for the edition number in the final output module. 1 is used if -e is not present. |
| -M=<size> | <size> indicates the number of pages (kbytes if size is followed by a K) of additional memory, c.link will allocate to the data area of the final object module. If no additional memory is given, c.link adds up the total data stack requirements found in the psect of the modules in the input modules. |
| -m | Prints linkage map indicating base addresses of the psects in the final object module. |
| -s | Prints final addresses assigned to symbols in the final object module. |
| -b=<ept> | Link C functions to be callable by BASIC09. <ept> is the name of the function to be transferred to when BASIC09 executes the RUN command. |
| -t | Allows static data to appear in a BASIC09 callable module. It is assumed the C function called and the calling BASIC09 program have provided a sufficiently large static storage data area pointed to by the Y register. |

This Page Intentionally Left Blank

# APPENDIX A

## DIFFERENCES BETWEEN RMA AND THE MICROWARE INTERACTIVE ASSEMBLER

There are some important differences between the Relocatable Macro Assembler (RMA) and the Microware Interactive Assembler (MIA), which is an "absolute" assembler. Some of the more important differences are:

1. RMA does not have an interactive mode. Only a disk file is allowed as input.

2. The output of RMA is a relocatable-object format file (ROF). The ROF file must be processed by the linker to produce an executable OS9 memory module. The layout of the ROF file is described later.

3. RMA has a number of new directives to control the placement of code and data in the executable module. Since RMA does not produce memory modules, the MIA directives "mod" and "emod" are not present. Instead new directives PSECT and VSECT control the allocation of code and data areas by the linker.

4. RMA has no equivalent to the MIA "setdp" directive. Data (and DP) allocation is handled by the linker.

This Page Intentionally Left Blank

# Appendix B

## Error Messages

When RMA detects an error, it prints an error message in the listing just before the source line containing the error. It is possible for a statement to have two or more errors, in which case each error is reported on a different line preceding the erroneous source line.

If the assembler listing is inhibited by the absence of the "-l" option, error messages and printing of erroneous lines still occurs. At the end of the assembly, the total number of errors and warnings are given as part of the assembly summary statistics. The error messages, erroneous source lines, and the assembly summary are all written to the assembler task's error/status path which may be redirected by the shell. For example:

```
rma sourcefile  o=sourcefile.o > src.error
```

Note that calling the assembler with the listing and object code generation both disabled by the absence of the " -l -o " options can be used to perform a quick assembly just to check for errors. This allows many errors to be found and corrected before printing of a lengthy listing. For example:

```
rma sourcefile
```

Sometimes the assembler will stop processing of an erroneous line so additional errors following on the same line may not be detected, so corrections should be made carefully.

Error messages consist of brief phases which describe the kind of error the assembler detected. Each error message is explained in detail in the list on the following pages.

## ERROR MESSAGES

Bad label

Incorrectly formed label found in label field.

Bad Mnemonic

Mnemonic was found in mnemonic field that was not recognized or was not allowed in the current program section.

Bad number

The numeric constant definition contains a character that is not allowed in the current radix.

Bad operand

Missing or incorrectly formed operand expression.

Bad operator

Incorrectly formed arithmetic expression.

Bad option

Unrecognized or incorrectly specified option.

Bracket missing

Can´t open file

A problem was encountered opening an input file.

Can´t open macro work file

A problem was encountered opening a macro work file.

Comma expected

Conditional nesting error

Mismatched if and else/endc conditional assembly directives.

Constant definition

Incorrectly formed constant definition.

DP section???

The comment field of the VSECT directive starts with "DP".

ENDM without MACRO

ENDR without REPT

Fail <message>

Fail directive encountered.

File close error

Incorrectly formed label found in label field.

Illegal addressing mode

The addressing mode cannot be used with the instruction.

Illegal external reference

External names cannot be used with assembler directives. If an operand expression contains an external name, the only operation allowed in the expression is binary plus and minus.

Illegal index register

The register cannot be used as an index register.

Label missing

This statement is missing the required label.

Macro arg too long

No more than 60 characters total can be passed to a macro.

Macro file error

Problem accessing macro work file.

Macro nesting too deep

Macro calls may be nested up to 8 levels deep.

Nested MACRO definitions

A macro cannot be defined inside a macro definition.

Nested REPT

Repeat blocks cannot be nested.

New symbol in pass two

See symbol lost.

No input files

An input files must be specified.

No param for arg

A macro expansion is attempting to access an argument that was not passed by the macro call.

Phasing error

A label has a different value during pass two than it did during pass two.

redefined name

The name appears more than once in the label field other than on a SET directive.

Register list error

The register names that are allowed in tfr, exg, and pul are: A, B, CC, DP, X, Y, U, S, and PC.

Register size mismatch

Both registers used in tfr and exg instructions must be the same size.

Undefined org

* (program counter org) cannot be accessed within a VSECT.

Unmatched quotes

Symbol lost?

Assembler symbol lookup error. The error could be caused by symbol table overflow or bad memory.

Too many args

No more than 9 arguments may be passed to a macro.

Too many object files

Only one "-o=" command line option is allowed.

Too many input files

A maximum of 32 files can be specified.

Value out of range

A byte expression value is less than -256 or greater than 256.

This Page Intentionally Left Blank

# APPENDIX C

## ASSEMBLY LANGUAGE PROGRAMMING EXAMPLES

The following pages contain two assembly language programming examples. They are:

UpDn - Program to convert input case to upper or lower.
LIST - File listing utility.

These programs are provided to give an example of what an assembly language program should be in the way of structure and form.

```
Microware OS-9 RMA - V1.0  83/07/07  13:14   updn.a      Page    1
updn - ASSEMBLY LANGUAGE EXAMPLE

00001               * This is a program to convert characters from lower to
00002               * upper case (by using the u option), and upper to lower
00003               * (by using no option).  To use type:
00004               *       updn u (for lower to upper) < input  > output
00005
00006                               nam   updn
00010
00011                               opt   l
00012                               ttl   ASSEMBLY LANGUAGE EXAMPLE
00013
00014 0010                 PRGRM    equ   $10
00015 0001                 OBJCT    equ   $01
00016
00017 00fa                 stk      equ   250
00018                               psect updn,PRGRM+OBJCT,$81,0,stk,entry
00019
00020 0000                          vsect
00021 0000                 temp     rmb   1
00022 0001                 uprbnd   rmb   1
00023 0002                 lwrbnd   rmb   1
00024 0000                          endsect
00025
00026 0000 a680            entry    lda   ,x+         search parameter area
00027 0002 84df                     anda  #$df        make upper case
00028 0004 8155                     cmpa  #'U         see if a U was input
00029 0006 2710                     beq   upper       branch to set uppercase
00030 0008 810d                     cmpa  #$0D        carriage return?
00031 000a 26f4                     bne   entry       no; go get another char
00032
00033 000c 8641                     lda   #'A         get lower bound
00034 000e b70002                   sta   lwrbnd      set it in storage area
00035 0011 865a                     lda   #'Z         get upper bound
00036 0013 b70001                   sta   uprbnd      set it in storage area
00037 0016 200a                     bra   startl      go to start of code
00038
00039 0018 8661            upper    lda   #'a         get lower bound
00040 001a b70002                   sta   lwrbnd      set it in storage
00041 001d 867a                     lda   #'z         get upper bound
00042 001f b70001                   sta   uprbnd      set it in storage
00043
00044 0022 30c90000        startl   leax  temp,u      get storage address
00045 0026 8600                     lda   #0          standard input
00046 0028 108e0001                 ldy   #$01        number of characters
00047 002c 103f8b          loop     os9   I$READLN    do the read
00048 002f 2519                     bcs   exit        exit if error
00049 0031 f60000                   ldb   temp        get character read
00050 0034 f10002                   cmpb  lwrbnd      test char bound
00051 0037 2507                     blo   write       branch if out
00052 0039 f10001                   cmpb  uprbnd      test char bound
```

```
00053 003c 2202              bhi    write       branch if out
00054 003e c820              eorb   #$20        flip case bit
00055 0040 f70000    write   stb    temp        put it in storage
00056 0043 4c                inca               reg 'a' standard out
00057 0044 103f8c            os9    I$WRITLN    write the character
00058 0047 4a                deca               return to standard in
00059 0048 24e2              bcc    loop        get char if no error
00060 004a c1d3      exit    cmpb   #E$EOF      is it an EOF error
00061 004c 2601              bne    exitl       not eof, leave carry
00062 004e 5f                clrb               clear carry, no error
00063 004f 103f06    exitl   os9    F$EXIT      error returned, exit
00064 0052                   endsect
```

```
Microware OS-9 RMA - V1.0  83/07/12  15:21   list.a          Page      1

00001           **********
00002           * LIST UTILITY COMMAND
00003           * Syntax: list <path>
00004           * List copies input from path to standard output
00005
00009
00010 0010               PRGRM    equ  $10
00011 0001               OBJCT    equ  $01
00012 00c8               STK      equ  200
00013
00014                             csect
00015 0000               IPATH    rmb  1         input path number
00016 0001               PRMPTR   rmb  2         parameter pointer
00017 0003               BUFSIZ   rmb  200       size of input buffer
00018                             endsect
00019
00020                             psect list,PRGRM+OBJCT,$81,0,STK,LSTENT
00021
00022 00c8               BUFFER   equ  200       allocate line buffer
00023 0001               READ.    equ  1         file access mode
00024
00025 0000 9f01          LSTENT   stx  PRMPTR    save parameter ptr
00026 0002 8601                   lda  #READ.    select read access mode
00027 0004 103f84                 os9  I$OPEN    open input file
00028 0007 2530                   bcs  LIST50    exit if error
00029 0009 9700                   sta  IPATH     save input path number
00030 000b 9f01                   stx  PRMPTR    save updated param ptr
00031
00032 000d 9600          LIST20   lda  IPATH     load input path number
00033 000f 30c900c8               leax BUFFER,u  load buffer pointer
00034 0013 108e0003               ldy  #BUFSIZ   max bytes to read
00035 0017 103f8b                 os9  I$READLN  read line of input
00036 001a 2509                   bcs  LIST30    exit if error
00037 001c 8601                   lda  #1        load st. out path #
00038 001e 103f8c                 os9  I$WRITLN  output line
00039 0021 24ea                   bcc  LIST20    repeat if no error
00040 0023 2014                   bra  LIST50    exit if error
00041
00042 0025 c1d3          LIST30   cmpb #E$EOF    at end of file?
00043 0027 2610                   bne  LIST50    branch if not
00044 0029 9600                   lda  IPATH     load input path number
00045 002b 103f8f                 os9  I$CLOSE   close input path
00046 002e 2509                   bcs  LIST50    ..exit if error
00047 0030 9e01                   ldx  PRMPTR    restore param ptr
00048 0032 a684                   lda  0,x
00049 0034 810d                   cmpa #$0D      end of param line?
00050 0036 26c8                   bne  LSTENT    ..no; list next file
00051 0038 5f                     clrb
00052 0039 103f06        LIST50   os9  F$EXIT    ..terminate
00053 003c                        endsect
```