MICROWARE C COMPILER USER'S GUIDE
APPENDIX D

RELOCATING MACRO ASSEMBLER REFERENCE

This appendix gives a summary of the operation of the "Relocating Macro Assembler" (named c.asm as distributed with the C Compiler). This appendix and the example assembly source files supplied with the C compiler should provide basic information on how to use the "Relocating Macro Assembler" to create relocatable-object format files (ROF). It is further assumed that you are familiar with the 6809 instruction set and mnemonics. See the Microware Relocating Assembler Manual for a more detailed description. The main function of this appendix is to enable the reader to understand the output produced by c.asm. The Relocating Macro Assembler allows programs to be compiled separately and then linked together, and it also allows macros to be defined within programs.

Differences between the Relocating Macro Assembler(RMA) and the Microware Interactive Assembler(MIA):

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

RMA output is an ROF file. 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.

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.

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

Symbolic Names.

A symbolic name is valid if it consists of from one to nine uppercase or lowercase characters, decimal digits or the characters "$", "_", "." or "@". RMA does not fold lowercase letters to uppercase. The names "Hi.you" and "HI.YOU" are distinct names.

Label 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 linked together). If no colon appears, the name will be known only in the PSECT in which it was defined. PSECT will be described later.

Undefined names.

If a symbolic name is used in an expression and hasn't been defined, RMA assumes the name is external to the PSECT. RMA will record information about the reference so the linker can adjust the operand accordingly. External names cannot appear in operand expressions for assembler directives.

Listing format.

```
00098 0032 59         +          rolb
00117 0045=17ffb8       label    lbsr  _dmove     Comment
^     ^    ^^         ^ ^        ^     ^           ^
|     |    ||         | |        |     |           Start of comment
|     |    ||         | |        |     Start of operand
|     |    ||         | |        Start of mnemonic
|     |    ||         | Start of label
|     |    ||         A "+" indicates a line generated by a macro
|     |    ||         expansion.
|     |    |Start of object code bytes.
|     |    An "=" here indicates that the operand contains an
|     |    external reference.
|     Location counter value.
Line number.
```

Section location counters.

Each section contains the following location counters:

```
     PSECT - instruction location counter
     VSECT - initialized direct page location counter
             non-initialized direct page location counter
             initialized data location counter
             non-initialized data location counter
     CSECT - base offset counter
```

Section directives.

RMA contains 3 section directives. PSECT indicates to the linker the beginning of a relocatable-object-format file(ROF) and

initializes the instruction and data location counters and assembles code into the ROF object code area. VSECT causes RMA to change to the data location counters and place any generated code into the appropriate ROF data area. CSECT initializes a base value for assigning offsets to symbols. The end of these sections is indicated by the ENDSECT directive.

The source statements placed in a particular section cause the linker to perform a function appropriate for the statement. Therefore, the mnemonics allowed within a section are restricted as follows:

These mnemonics are allowed inside or outside any section: nam, opt, ttl, pag, spc, use, fail, rept, endr, ifeq, ifne, iflt, ifle, ifge, ifgt, ifp1, endc, else, equ, set, macro, endm, csect, and endsect.

Within a CSECT: rmb.

Within a PSECT: any 6809 instruction mnemonic, fcc, fdb, fcs, fcb, rzb, vsect, endsect, os9 and end.

Within a VSECT: rmb, fcc, fdb, fcs, fcb, rzb and endsect.

PSECT directive.

The main difference between PSECT and MOD is that MOD set up information for OS-9 and PSECT sets up information for the linker(c.link in the C compiler).

PSECT {name,typelang,attrrev,edition,stacksize,entrypoint}

name — Up to 20 bytes (any printable character except space or comma) for a name to be used by the linker to identify this PSECT. This name need not be distinct from all other PSECTs linked together, but it helps to identify PSECTs the linker has a problem with if the names are different.

typelang — byte expression for the executable module type/language byte. If this PSECT is not a "mainline"(a module that has been designed to be forked to) module this byte must be zero.

attrrev — byte expression for executable module attribute/revision byte.

edition — byte expression for executable module edition byte.

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

entrypoint word expression entrypoint offset for this PSECT. If the PSECT is not a mainline module, this should be set to zero.

PSECT must have either no operand list or an operand list containing a name and five expressions. If no operand list is provided, the PSECT name defaults to "program" and all other expressions to zero. There can only be one PSECT per assembly language file.

The PSECT directive initializes all counter orgs and marks the start of the program module. No VSECT data reservations or object code may appear before or after the PSECT/ENDSECT block.

Example:

```
psect myprog,Prgrm+Objct,Reent+1,Edit,0,progent
psect another_prog,0,0,0,0,0
```

VSECT directive.

VSECT {DP}

The VSECT directive causes RMA to change to the data location counters. If DP appears after VSECT, the direct page counters are used, otherwise the non-direct page data is used. The RMB directive within this section reserves the specified number of bytes in the appropriate uninitialized data section. The fcc, fdb, fcs, fcb and rzb (reserve zeroed bytes) directives place data into the appropriate initialized data section. If an operand for fdb or fcb contains an external reference, this information is placed in the external reference part of the ROF to be adjusted at link or execution time. ENDSECT marks the end of the VSECT block. Any number of VSECT blocks can appear within a PSECT. Note, however, that the data location counters maintain their values between one VSECT block and the next. Since the linker handles the actual data allocation, there is no facility provided to adjust the data location counters.

CSECT directive.

CSECT {expression}

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

RZB statement.

Syntax: RZB <expression>

The reserve zeroed bytes pseudo-instruction generates sequences of zero bytes in the code or initialized data sections, the number of

which is specified by the expression.

COMPARISON BETWEEN ASSEMBLY PROGRAMS FOR THE MICROWARE INTERACTIVE ASSEMBLER AND THE RELOCATING MACRO ASSEMBLER

The following two program examples simply fork BASIC09. The purpose of the examples are to show some of the differences in the new relocating assembler. The differences are apparent.

```
* this program forks basic09
          ifp1
          use ..../defs/os9defs.a
          endc

PRGRM     equ $10
OBJCT     equ $01

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

name      fcs /basic09/
prm       fcb $D
prmsize   equ *-prm

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

MACRO INTERACTIVE ASSEMBLER SOURCE

```
          ifp1
          use defsfile
          endc

          mod siz,prnam,type,revs,start,size
prnam     fcs /testshell/
type      set prgrm+objct
revs      set reent+1

          rmb 250
          rmb 200
name      fcs /basic09/
prm       $D
prmsize   equ *-prm  (continued)
```

```
size    equ .
start   equ *
        leax name,pcr
        leau prm,pcr
        ldy #prmsize
        lda #PRGRM+OBJCT
        clrb
        os9 F$FORK
        os9 F$WAIT
        os9 F$EXIT
        emod
siz     equ
```

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. The 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 subroutines 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.

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 redefine the 6809 instructions (LDA, CLR, etc.) themselves by defining macros having identical names. 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 that is 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.

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 to make 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
```

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...
```

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.