McDonnell Douglas Field Service Company

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE PROGRAMMING AND ARCHITECTURE

Garrett D. Hildebrand / Robert W. Murray

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE PROGRAMMING AND ARCHITECTURE
- CLASS NOTES PART III -

I. MODE FORMAT

The overall format of a mode can exist in three different ways. One format would be the mode consisting of the source code only, the second would consist of the object code only, and the third format consisting of both source code and object code.

The first of the above mentioned formats is one you will first use when constructing an assembler routine. Until the mode is assembled for the first time, it will contain only source code. The object code generated at assembly time can be removed to create a source-only format once again via the STRIP-OBJECT verb. Source code format will be discussed in detail in the following section.

The second mentioned format is object code only. This type of format is seen in the SYSTEM-OBJECT file. This type of format is generated using the STRIP-SOURCE verb. Each line of an item of this type consists of a sub-value mark (X'FC'), followed by a three-digit hex displacement, followed by the literal character ' ', followed by the object code. Example:

```
007 \000 201E0210100A
```

The other format of a mode would be one that contains both source and object. This type is descriptive of the mode once it has been assembled. Since the editor has special commands for use with assembly language programs, it is not necessary the strip the object code for subsequent editing.

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE
PROGRAMMING AND ARCHITECTURE
- CLASS NOTES PART III -

II. STATEMENT LINE FORMAT

Each statment line in a mode has a particular format which is recognized by the assembler. This format uses the the system delimiters VM (X'FD') and SVM (X'FC') to separate the various seqments of the mode. This format is as follows:

Source VM Macro {VM Macro} SVM Object {SVM Object} VM Errors

This type of format will NOT always be the same. It will vary depending on a number of reasons. Obviously, if you haven't assembled your mode, or if you used the STRIP-OBJECT verb, then the source line will be all that you have and no other delimiters will be present. Your state- ment lines will will look like this:

```
045  ZERO R14WADSP DETACH REGISTER
```

If the opcode is a primitive, then there will be no macro statements in the line. A sub-value mark and the generated object code will follow the source statement.

```
016  MOV D0,D1\021 F0048006
```

If the opcode is a macro, then is will follow the source statement separated by a value mark. This is followed by a sub-value mark and the generated object code.

```
014  B !MD999] B: !MD999\001 1E6E
```

Errors are only appended to the line if an error has occurred during the assembly process. The following is an illegal statement an thus get flagged as such.

```
025  MCC C'?',T0]*ERR:  ILGL OPCD:  MCCNT
```

If you are looking at an item in SYSTEM-OBJECT or a mode that has had the source code stripped, then the lines will begin with a SVM followed by the object code.

```
006 \000 021E0210100A
```

In summary, any of these values or a combination of these values can be null. The assembler uses the system delimiters to separate the existing values and to locate fields for formatted listings.

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE
PROGRAMMING AND ARCHITECTURE
- CLASS NOTES PART III -

III. SOURCE LANGUAGE FORMAT

Source language is stored in an item, with one statement or source line per attribute. Each source line begins with a label field. The label field is followed by an opcode field, an operand field, and a comment field. Each of these fields is separated from the other with a blank. Since the assembler recognizes a blank as a delimiter for the fields, it cannot be used within the label field, opcode field, or operand field. The exeption is when it in contained in a literal string enclosed in single quotes. Example:

LABEL FIELD OPCODE FIELD OPERAND FIELD COMMENT FIELD

The exception to this is the first 6 attributes of the source mode, which are assumed to contain the following:

```
001  FRAME statement
002 *Classification      (major class of software in this mode)
003 *Revision-date       (changed by assembler @ each assembly)
004 *Revision-level      (such as '04' for revision '4')
005 *Modification-level  (such as '10' for mod '1' as in 04.10)
006 *Changed-by          (such as 'GDH' or 'RWM')
```

The FRAME statement begins with the literal character ' ', followed by the literal string 'FRAME', followed by the literal character ' ', followed by a number between 1 and 4095, representing the ABS (ABSolute area) Frame-id (FID), as in ' FRAME 478'.

Lines 2 through 6 begin with the literal character '*', followed by textual information as mentioned above. Notice that the revision date on attribute 3 will be automatically inserted after each assembly.

The format of these first six lines is a convention used by all system programmers when writing assembly language on any of the systems. There is an 'X' option that can be used with the AS verb at assembly time that will instruct the assembler to suppress the automatic dating and assemble every line.

IV. LOCATION AND SEPARATION OF FIELDS

The Label Field must begin in column one of a source line. The literal character ' ' is used as a delimiter for the label field, opcode field, and operand field. If a space appears in column one of a source line, the label field is null. As is used in lines two through six of a mode, the literal character '*' appearing in the Label Field causes the assembler to treat the entire line as a comment, and spaces are then ignored.

1 LABEL FIELD

The Label Field cannot contain the characters '+', '-', ''', and '='. Also, the character '*' may not be embedded in a label, although it may be used in place of a label to denote a comment line. Use of 10 or more characters will spoil the cosmetics of the assembly source/object listing produced with the MLIST verb or with the EDitors 'AS' option by shifting the remainder of the statement line to the right. The effective maximum in this field is 14. The label field is used to define labels for a particular location within a mode (used for code locations or data definition locations), or to store a symbol name for a symbol definition statement. Additionally, the label field can be optionally used with most assembler directives.

2 OPCODE FIELD

The opcode field, also called the operation field, is separated from the label field by a single space (' '). Valid opcodes are either defined in the OSYM file as primatives (REAL machine instructions, e.g., 'B' for branch), macros (combinations of primatives, also defined in OSYM, e.g., 'MBD' for Mov Binary to Decimal), or assembler pseudo-opcodes, also known as assembler directives (e.g., ORG, for Origin). The maximum limit of characters in the opcode field without shifting the remainder of the source line is five.

3 OPERAND FIELD

The operand field entries and expressions are optional; this is usually determined by the preceding opcode. Entries in this field are separated by commas. Entries in this field may be constants, symbols and expressions. The characters '+', '- ', '=', and '*' have special meaning, as described below. Except for literal strings embedded between the ' character, spaces are not allowed. The maximum limit of characters in this field that will not shift the comment field is seventeen.

- Operand Field Location Counter Symbol

The asterisk (*) is used to represent the current contents of the location counter, which is assumed as R1. The location counter is maintained by the assembler. Unless altered by the ORG directive, the counter begins at one and increments by the number of bytes used to store machine code generated by the assembly or data embedded in the mode. The range of values returned will be no less than zero and no more than 511.

- Other Operand Field Special Symbols

The mathmatical operators + and - may be used in two ways. They may be used preceding a constant to indicate sign, or they may be used after the location counter symbol as an operator after the location counter symbol * (- causes twos complement of the location counter), or they may be used to combine text in a symbol name as these are valid symbol characters.

- Constants

A constant is a string of characters which imply value by their form. Three types of constants are recognized:

decimal
hexadecimal
character

Decimal constants can be from one to ten digits between -4294967296 and 4294967295. A decimal constant can be preceded by the + and - signs.

Hexadecimal constats can be from one to eight digits which are enclosed in single quotes and preceded by an X. Optionally, the X may be preceded by the + and - signs. Examples are:

X'FFC' -X'FFAC' +X'ACE'

Hexadecimal constants may range up to a magnitude of X'FFFFFFFF'.

A character constant is a string of characters enclosed in single quotes and preceded by a C. Any characters other than X'0D', X'0A', and the system delimiters may be used in the string. For example, C'this is the time for a beer'. Double quotation marks have no special meaning in the string.

o Expressions

The assembler evaluates expressions that use the operators + and - . Expressions are used in the operand field, and its terms can be symbols, constants and the location counter symbol (*).

The allowed combinations of operators and terms are as follows:

```
[+ or -]symbolname

*[{+ or -}constant]...

[+ or -]constant[{+ or -} constant]...
```

Blanks are not allowed between the terms of an expression. Some valid expressions are:

```
SYM1        *1          +X'FF'
+SYM2       *1+1        -X'FD'
-ZSYM       C'BYTE'     *-
```

When an expression is evaluated, it must result in a value that fits in 32 bits. If it does not, the value is truncated and an error message is printed.

Some sequences that might give valid but unexpected results are:

SEQUENCE	RESULTS
SYM1+SYM2	treated as the symbol name 'SYM1+SYM2' since the + or - can be used in a symbol name.
6+SYM1	the +SYM1 is ignored, yielding a 6 only.
*+SYM1	the +SYM1 is ignored, yielding the value of the location counter only.
*-SYM1	the SYM1 is ignored, but the twos complement of the location counter results.
-*	the value of the location counter only - results, not the twos complement.

4 COMMENT FIELD

Any commentary information (any text) may be included in this field, which follows the operand field and is separated from it by a space. Reasonable comments should be on most lines of code with executable operators, and most label lines. Grammar should be sacrificed to shortness.

IV SYMBOLS

A symbol is the name of a location (e.g., LOCALLAB EQU *, where LOCALLAB refers to the location counter at the point the equate was issued), a register (e.g., OBUF AR 11, causing OBUF to be another name for R11, or the OB register), or a value assigned via one of the directives in a definition statement). The value given to a symbol is not implied by the symbol itself, but must be assigned to that symbol by its appearance in an assembly statement or by an item in the PSYM file. file.

A symbol can contain any characters except the following:

Line Feed X'0A'
Return X'0D'
comma c','
apostrophe
blank X'20' (space)
any system delimiter X'FB' - X'FF'

A symbol cannot begin with a numeric character, but may contain them, nor can it begin with a plus sign, minus sign, equal sign or asterisk. The effective limit of a symbol is 14 characters.

V. OPERAND FORMATS

When specifying an operand field entry in an instruction statement (as opposed to an assembler directive statement), the entry will typically be defined by an item in either the PSYM or TSYM files. Items in the TSYM file are created by the assembler in response to literals expressed in an operand, or by assembling data definitions created with assembler directives. In this latter situation, storage locations have been defined by the user specifying a symbol that represents the location and size of the storage-location.

Some additional operand formats are provided that allow another method of specifying a storage-location.

o SPECIAL OPERAND FORMATS

A typical operand field entry will be defined in either PSYM or TSYM. The definition in one of these files contains a symbol code (attribute one), a displacement off the base register (attribute two), and a hex number between the value of X'0' and X'F' that specifies the base register number attribute three).

The assembler translates an instruction like

```
MOV D1,D0
```

by looking up the symbols D1 and D0 in PSYM, thus aquiring the symbol codes for the two operand entries, then looking up MOVDD in the OSYM file.

Operand entries may specify immediate data via a constant, or a register via a symbol defined with the R symbol code, or a storage location via a symbol defined with the B, C, H, T, D, S, or F symbol codes, as in the D1,D0 example above.

Another way is to use the special operand formats that allow specification of the symbol code and displacement directly in the operand; this is done by appending this information to a base register symbol (e.g., such as R3, defined by PSYM) with a semicolon. THE OFFSET IS DIFFERENT THAN WITH A STANDARD SYMBOL. While a standard symbol's symbol code attribute has an implied displacement in units of bits, bytes and words (double byte), the special operand format employing the semicolon implies offsets of up to 6 byte-units. Operand lengths are the same.

The format is as follows:

Rn;Sm

Where 'Rn' refers to one of the 16 register symbols R0 through R15, 'S' refers to the symbol code employed, and 'm' refers to the number of units of displacement, and is used to calculate offset, from the base register specified by 'Rn'. Examine the following table:

FORMAT	RELATIVE DISPLACEMENT	OPERAND LENGTH
Rn;Bm	m Bits off Rn	1 Bit
Rn;Cm	m Bytes off Rn	1 Byte (DEFC)
Rn;Hm	m Bytes off Rn	1 Byte (DEFH)
Rn;Tm	m words off Rn	2 Bytes (DEFT)
Rn;Dm	(2*m) words off Rn	4 Bytes (DEFD)
Rn;Fm	(3*m) words off Rn	6 Bytes (DEFF)
Rn;Sm	(3*m) words off Rn	6 Bytes (DEFS)

EXAMPLES:

1) MOV MBASE,R10;D4 — Move double-tally MBASE to a double-tally destination at byte 15 relative to R10 (16 bytes past the byte addressed by R10).

```
R
1
0
<D0><D1><D2><D3><D4>etc...
0123456789ABCDEF0123456789ABCDEF
0               1
```

2) MCC R0;C2,R15 — Move character 2 of the PCB (PRMPC) to the byte address by R15. Recall that R0 points to byte zero of the Primary Control Block.

3) SB R0;B128 — Sets bit X'80' of the PCB, which is bit zero of byte X'10' (AFLG).

Note that the notation R10;D4 does not imply two operand entries. It is one entry which specifies a base address (R10), a symbol code (D), and a displacement (8 words, or 16 bytes).

Note that the use of the 'Rn;Sm' format does not allow you to Express a value greater than 255 for 'm' if symbol code 'S' is either an 'H', 'C', or 'B'. If this restriction is abused, the assembler gives the message

```
*ERR: TRUNC
```

and enters the editor.

Also note that the 'Rn;Sm' format does not cause the assembler to create items in the TSYM file. Instead, the information is translated directly into the object code. For example, compare a type of definition to the special operand format:

```
1)   PCB#D4  DEFD   0,X'8'     causes an item called 'PCB#D4'
                               to be created in the TSYM file.

             MOV    PCB#D4,D0  causes a lookup of item
                               'PCB#D4' in the TSYM file. It
                               has the format

                                   PCB#D4
                               001 D
                               002 008
                               003 0          <-- OR NULL
```

The object F0068008 is generated, where the '008' is created from attribute three and two, respectively, of the TSYM item PCB#D4.

```
2)           MOV    R0;D4,D0   causes no lookup in TSYM. The 4
                               following the symbol code 'D'
                               is multiplied by 2 to create
                               the displacement (in words) off
                               base register R0.
```

The object F0068008 is also generated by this last example, even though no lookup has taken place in TSYM.

SPECIAL NOTE ON THE USE OF THE 'Fn;Sm' FORMAT

Use of this format is restricted to the operand field of INSTRUCTION statements, and is not used with the assembler DEFk type directives.

VI. DATA GENERATION DIRECTIVES

Data generation directives are used in place of an instruction mneumonic to direct the assembler to generate data for use by the assembly mode. Each datum may be named symbolicaly, thus allowing definition of symbols. These symbols can then be used in the operand field as entries in instruction statements.

Data that are generated by means of these directives will reside in the mode, and take up space in the MLOADed ABS frame. Since the assembler assumes a base register of R1 (recall that R1 effectively points to byte zero of the current ABS frame during execution of any instruction), all data that require a word displacement (tallys, doule-tallys, triple tallys and storage) are aligned on an even byte boundry, periodically wasting a byte in the frame.

The maximum displacement that ANY instruction can hold is 255, or X'FF'. Hence, byte data (half-tallys and characters) must appear in bytes 0-255 of the mode/frame if they are to be referenced.

Multi-word data (double-tallys, triple-tallys, storage registers) that cross the frame boundry can be referenced by word-oriented instructions (e.g., MOV, LOAD) without error as assembly time; however, this situation will cause an error at execution time, so be careful of this.

- INCLUDE directive

Allows a set of externally define DEFS to be included.

FORMAT: INCLUDE item-name

This will allow a series of DEFS to be included during the assembling of a mode that are defined in another item in the file. The DEFS mode must be define in the same file as the item being assembled, and must be assembled itself.

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE
PROGRAMMING AND ARCHITECTURE
- CLASS NOTES PART III -

- ALIGN directive

Aligns the locations counter on an even byte boundry.

FORMAT: [symbol] ALIGN

If a symbol is present in the label field, it is defined in TSYM with the L symbol code. There is no practical purpose to using the optional symbol in the label field, just that the assembler syntax allows this. Align is rarely used, since most of the data generating instructions align anyway.

- CMNT directive

Allows a comment to be appended to a label.

FORMAT: [label] CMNT [operand] [comment]

This is little used, as most people us the asterisk in column one of a source line to denote a comment line. None the less, it may be employed to provide a comment line at the label for a routine or subroutine. It's operand is ignored by the assembler, however, the assembler will print the first string of characters after the opcode field in the operand field.

To avoid a messy display, it is recommended that an asterisk * be used in the operand field, so the comment is properly aligned in the listing.

If the label is employed, the assembler will generate an item for it in TSYM with an L symbol code, the current location counter value, and the base register R1. An example of how CMNT may be used follows

```
                B     CXDTA

                  o
                  o
                  o

        CXDTA     CMNT  *        CONVERT HEX TO DEC ASCII
```

- CHR directive

Generates character data on byte in length.

FORMAT: [symbol] CHR expression

If a symbol is present in the label field, it will be defined with the C symbol code, the current location counter value, a base register = R1, and the current value of the operand expression, which is placed on attribute four of the TSYM item (for used with immediate data instructions not requiring the register/offset).

```
EXAMPLE:   SM  CHR  X'FF'
           SPA CHR  C' '
```

- DTLY directive

Generates an aligned double word of data 32 bits in length.

FORMAT: [symbol] DTLY expression

If a symbol is present in the label field, a TSYM item will be created with a D symbol code, the current word location after alignment, and the base register R1.

```
EXAMPLE:   =D1000  DTLY  1000
```

Generates X'000003E8' double tally at current location counter (after alignment).

```
           DBLONES DTLY  -1
```

Generates all ones doubly tally X'FFFFFFFF'.

- EJECT - Eject the page

FORMAT: [symbol] EJECT

This directive is not used by the assembler unless the optional (and useless) symbol is defined. The statement is passed to the MLIST processor which responds by outputting a form feed. Used for cosmetic purposes. For example:

```
           o
           o
           ENT   MD999   WRAP UP MODE
           EJECT
```

- END - End of listing

FORMAT: [symbol] END

This directive is not used by the assembler unless the optional symbol is defined. It is used only as a visual aid that the end of the mode has been reached. Cosmetic only.

- HTLY directive

Generates half word data 8 bits in length.

FORMAT: [symbol] HTLY expression

If the symbol is present in the label field, it will be defined in TSYM with the H symbol code, the current location counter value, and base register R1.

```
EXAMPLE:   ACE  HTLY  1
            HTLY  X'0B'
           =H7  HTLY  7
```

- TLY directive

Generates an aligned word of data 16 bits in length.

FORMAT: [symbol] TLY expression

If a symbol is present in the label field, a TSYM item will be created with a T symbol code, the current word location after alignment, and the base register R1.

```
EXAMPLE:   =T32267  TLY  32267
```

Generates X'7E0B' tally at current location counter (after alignment).

```
           TMASK    TLY  -1
```

Generates all ones tally X'FFFF'.

- MTLY directive

Generates modal address (16 bits).

FORMAT: [symbol] MTLY expression1,expression2

If the symbol is present in the label field, it will be defined in TSYM with the M symbol code, the hex equivalent of expression1 on attribute 2, and the hex equivalent of expression2 on attribute 3. Also, an item in TSYM will be created defining the location of the tally generated. It will have the id symbol:.

This directive is used to generate tally containing a mode-id in the modal address format, where the first four bits represents the modal entry number (derived from expression1, in the range of X'0'-X'F'), and the low-order 12 bits represents the FID in which the mode is stored, which is derived from expression2.

Although the symbol is optional, it is always employed. Note that this directive is not usually used to define a mode id for the purpose of using the symbol in the operand field of either the ENT or BSL instructions. It is generally employed so that XMODE, RMODE, WMODE or CMODE can be loaded with the generated word, although a symbol defined with the DEFM directive may also be employed here as well.

EXAMPLE: DDMODE MTLY 0,493

Generates the tally X'01ED'.

- TEXT directive

Generates binary equivalent of character of hex strings.

FORMAT: [symbol] TEXT string[,string]...

The TEXT directive will assemble character or hexadecimal strings into the mode that is being assembled. The character and hex strings are represented using the C and X characters prior to the string and surrounded with single quotes. Any number and combination or C and X literals separated by commas is permitted.

o ORG directive: Asserting control over the location counter.

Used to set the location counter to a different value.

FORMAT: [symbol] ORG expression

This directive sets the location counter to the value of the expression. It may also be used to assign an offset off R1 to a label, if a symbolis employed in the label field. The ORG statement can be used to create a mode that will patch a specific location of another modes ABS frame; it can be used to generate the equivalent of the VS directive; and it can be used to reserve space for DEFk fields, if no data generation is required, and the base register in the DEFk directive is R1.

Some Examples of the use of ORG:

1.) SVL#1 ORG * causes an item to be generated in TSYM with a symbol code of 'L', the current location counter value, and a base register of R1.

2.) ORG SLV#1 causes the location counter to be reset to the value of the symbol SVL#1.

3.) ORG *+1 increments the location counter by one, thus causing a byte of the target ABS frame to be unaltered

Other examples of the use of ORG:

4.)

```
        ALIGN
SAVR4   DEFS  1,*    DEFINE STORAGE REGISTER AT CURRENT
                     LOCATION COUNTER OFF R1.
        ORG   *+6    INC LOCATION COUNTER BY LENGTH OF
                     SAVR4 STORAGE REGISTER.
```

5.) TO SET UP A 8 BIT FLAGS FIELD, 'ORG' IS A HELP...

```
SBITS  DEFH  1,*     DEFINE 8 BIT HTLY AT CURRENT LOCATION
                     COUNTER OFF R1.

SBIT0  DEFB  1,*     DEFINE BIT ZERO OF SBITS HTLY

SBIT1  DEFB  1,*+1   DEFINE  BIT  ONE 1  BIT  PAST CURRENT
                     LOCATION COUNTER  (CURRENTLY POINTING
                     AT LOCATION OF BIT ZERO OF SBITS).

SBIT2  DEFB  1,*+2   DEFINE BIT TWO OF SBITS HTLY

SBIT3  DEFB  1,*+3   DEFINE BIT THREE OF SBITS HTLY

SBIT4  DEFB  1,*+4   DEFINE BIT FOUR OF SBITS HTLY

SBIT5  DEFB  1,*+5   DEFINE BIT FIVE OF SBITS HTLY

SBIT6  DEFB  1,*+6   DEFINE BIT SIX OF SBITS HTLY

SBIT7  DEFB  1,*+7   DEFINE BIT SEVEN OF SBITS HTLY

       ORG   *+1     INC LOCATION  COUNTER BY  ONE TO SKIP
                     BYTE DEFINED BY HTLY SBITS

       o
       o
       o

       ZERO  SBITS   INITIALIZE SBITS HTLY TO ALL ZEROES.
```

6.) TO EMBED YOUR INITIALS AT THE END OF THE ABSFRAME...

```
      ENT   MD999
      END   *       END OF LISTING
*
* EMBED AUTHORS INTITIALS AT END OF ABSFRAME
*
      ORG X'1FA'
      TEXT C' *RWM'
      EJECT
```

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE
PROGRAMMING AND ARCHITECTURE
- CLASS NOTES PART III -

III. THE STRUCTURE OF SOURCE CODE

INTRODUCTION

This section of the handout deals with the overall structure of code in a mode. Being exposed to the various directives and symbolic instructions presented prior to this, a lesson is at hand concerning the way in which the format of code should be laid out.

Although the explanations herewithin are not manditory for writing code, they present a guideline for a general format. The raison d'etre of this section is that source code should be easy to read, understand and manipulate. Of course, everybody is best off writing code in a way that's most comfortable to them, so the first draft should not meet any special format criteria. However, as the code settles into a more permanent form, it should be edged towards a more standard format.

INTRODUCTION TO REALITY ASSEMBLY LANGUAGE PROGRAMMING AND ARCHITECTURE
- CLASS NOTES PART III -

THE **FRAME** STATEMENT

The FRAME statement goes on line one of the source code item. This is required by the assembler or an error message will result. Nothing more needs to be said.

THE **COMMENTS.**

The standard comment lines 2 - 7 are next with the appropriate information contained in them as outlined in the section on SOURCE LANGUAGE FORMAT. Follow this section with a comment line as well.

THE **INCLUDE** BLOCK

Any external DEFS that will be INCLUDEd in the assembly of the mode should be entered here. This can be spotted at the top of the frame and alert anyone reading the code that there are references in the instructions that are not defined in the mode and not contained in PSYM. One can then aquire the DEFS mode for reference, if desired. This block should be followed by an "*" line if present, else null.

THE **ORG 0 ; CHR cc ; {LABEL EQU R1}**

If it is desired to define byte zero of the frame, then it is done here. Although it can be done elsewhere in the frame, this is where it should occur so it is not necessary to save the program counter and restore it back again. The LABEL EQU R1 should be used in any case where an N-type operand is useable (i.e. BCE LABEL,R15,DEST). The LABEL should be used in any instructions referencing it, rathar that R1.

THE ENTRY POINT GROUP.

Entry points are simply a branch to the code which does the idea implied by the entry point. Entry points should not have labels on them. They are not branched to internally, and should not look like it.

The label to which an entry point branches should be the logical name of the routine preceeded by an exclamation point ("!"), ALWAYS! No other labels should start with or contain an exclamation point. This convention will mark those labels as being externally addressable versus the labels that are internally branched to.

The entry points comment field should always contain the number of the entry point, followed by a short explanatory comment which indicates the use of the entry point.

It would be preferrable if the remaining entry points were filled with the TEXT X'0808' directive, if space allows. This would cut down the risk of destructive execution by error. If so, the comment should aslo contain the entry point number, followed by the text "AVAILABLE".

The entry point group should be followed with an asterisk.

Since an entry point is a branch to a segment of code designed to carry out a specific function, it would be convenient to use another opcode which would be a synonym of the "B" opcode as far as the assembler is concerned, to act as the entry point operator. The symbol "EP" (Entry Point) should be used in place of the "B", and the symbol "ILEP" (ILlegal Entry Point) should be used in place of the TEXT X'0808' currently used. However, these symbols are not contained in the OSYM and thus would not assemble unless you created them yourself. Hopefully they will be included in future releases of OSYM.

DEFINITION OF EXTERNAL DATA.

This block should start with the DEFMs, sorted by frame and entry point. The labels naming the entry points should be the same as the referenced entry point's breanch address, without the exclamation point. This block should be followed by an "*", if there are any.

Then the DEFXs that define data should be next. These should be sorted by register. These should also be followed by an "*" if any exist.

DEFINITION OF LOCAL DATA.

These are the XTLYs, ADDRs, and DEFXs that define local data. These should be in order by size, but can be mixed for the sake of saving space. Its handy to start with a half-tally to take advantage of the slack byte. This should be followed by an "*" line.

TEXTURAL PROLOG AND CODE SEGMENT.

A short but reasonably explanatory description of the first section of code should follow. It should preceed the label to which the entry point branches, and be followed by an "*" line.

Following is the entry point label and the code segment.

The rest of the mode should follow the pattern:

```
TEXTUAL PROLOG
ENTRY POINT LABEL
CODE SEGMENT
```

until the end of the frame where the epilog should be made up of the sequence: "*" line; END; "*" line. This would make finding the end of the code segment easier, and give a couple of lines of slack before the literal pool, which tends to push code off the top ofthe screen.