*Machine Language Assembler for 2200*

## Index

*Index - Continued*

## Introduction

The Machine Language Assembler for the Wang 2200 system was written to facilitate the generation of code required to run the 2200 computer. The Assembler takes Mnemonic code from either a disk file created by the Editor program, or code manually entered on the keyboard. In turn, the input code is evaluated and turned into machine compatable form.

The Machine Language Assembler, referred to as **ASSEMBLE**, is a two pass assembler. The first pass evaluates Control and Data memory addresses and assigns locations to the symbology. The second pass of **ASSEMBLE** results in the production of both an Object (Binary load) file, as well as a printable Assembled listings.

**ASSEMBLE** supports a substantial error checking capability as well as well as many control pseudo-ops for flexibility in programming. Macro assembly is permitted by utilizing the disk as a library of 'Macro' routines that may be invoked during the assembly stages.

Prior to proceeding with this manual, it may behoove the user to have read the 'Wang 2200 Instruction Set' document by Computer Concepts Corporation.

## Loading ASSEMBLE

The ASSEMBLE program is loaded from the host disk by entering the following line:

LOAD RUN "ASSEMBLE"(return key)

When loaded, the following messages will appear on the screen:

```
Wang Machine Language Assembler       Version 3.0 ###############
                                                  ## Computer ##
                                                  ## Concepts ##
                                                  ###############
Enter input file please: ?
```

The user must enter the name of the source text file. If that file does not exist, the system will display an error message and request the input file again. Only if ASSEMBLE finds the source file will the next question be asked.

Alternatively, the user may elect no input file. That is to say that the input will be from the keyboard. This is useful when testing small programs without having to run through the editor. This feature is invoked by simply entering a RETURN key to the file name prompt.

If a file was entered, and the system did find the program, the following is requested:

```
Output list device  005
```

The ASSEMBLE is preset to display the assembled listing during Pass 2 to the CRT console. If the user wishes to have a listing made to the printer, simply type 204 or 215 to the response.

```
Enter output file: ?
```

The Object (Binary) can be sent to a file on the disk for processing by the PATCHER programmer by entering the name of the file to the prompt. If no object file is to be generated, press a RETURN key here.

ASSEMBLE verifies that no program already exists by that name and proceeds with the assembly. Errors in the source will result in diagnostic messages appearing on the listing device during Pass 2. An assembly listing example, of the Bootstrap Proms, is included in the Appendix of this manual for reference.

If no errors in the source file are detected, the object file, if created, may then be either loaded or merged with another file.

## 2.0 Proceesing Syntax and Format of Input lines

In general, the structure of the Assembler follows the generally accepted style of machine language assemblers for Minicomputers. Certain licenses were taken to acheive a balance of speed and versatility within the limitations of Wangs' BASIC operating system.

The object of this section is to familiarize the reader with the various mathematical processes available, as well as develop the syntax required for the machine mnemonics.

A rather loose discipline as regards to format of the input line has been programmed in. ASSEMBLE is quite forgiving if one uses to many spaces to seperate datums.

For sake of readability of the code, profuse comment fields are recommended. Unlike BASIC, comments only take up room on the source file, not the final product.

The ASSEMBLER requires as input a BA type format file. Each line is terminated by an $0D character. Multiple spaces may be compressed into a TAB character,$7E, for further reduction in file space, as well as making a readable listing file.

This section will describe the basic math operators permitted, as well as rules for symbology and syntax of command lines.

The mnemonics themselves are described in the "Wang 2200 Instruction Set" document produced by Computer Concepts Corporation. The reader should be at least familiar with the basic machine level set prior to proceeding with this section.

## 2.1 Mathematical Operators

All expressions associated with Pseudo Op codes, as well as normal machine opcodes, may contain mathematical expressions. The expression is evaluated with logic similar to Reverse Polish Notation (RPN). Operators recognized by the assembler are the following:

| | |
|---|---|
| -- | Equivelent to + |
| ++ | Same as + |
| + | Addition |
| - | Subtraction |
| / | Division |
| * | Multiplication |
| ↑ | Raise to power n |
| ! | Logical OR data |
| & | Logical ANDing of data |
| () | Nesting levels |

Furthermore, the following are of special meaning:

| | |
|---|---|
| * | Current location Counter of Data or Control memory. Valid only in first position of Numeric evaluation. |
| . | Current location of Data or Control memory Valid at any place in the Numeric evaluation sequence. |
| % | Binary Data |
| $ | Hexidecimal Data |
| ' | Data following is Ascii Text. Must terminate with '. |

Symbology cannot contain any mathametical operator. The RPN processor will split the symbol into more than one symbol at the operator, causing invalid references. Symbology should not have their first characters equal to any of the special characters either. Some examples of invalid symbols are:

```
'LOCATION      EQU    $5000
```

The Assembler will evaluate this correctly. However, attempts to use the symbol 'LOCATION will result in an error, because the Symbol processor will believe that this is supposed to be text, rather than a symbol.

```
LOCATION'      EQU    $5000
```

The above is legal, as long as symbol does not start with '.

```
LOC+500!       EQU    $6000
```

Illegal symbol because of mathametical operators.

## 2.2 Math Operator precedence

The lexical scanner employed within the RPN processor will sub group all expressions by the following order:

```
(          Invokes precedence
)          Ends precedence
↑
/,*
-,+
!,&
```

As an example, the expression 80/2*5 will result in the value 200. By using the parenthesis to change precedence, the expression 80/(2*5) will result in the value 8.

All parenthesis, if used, must be balanced, else an error message will be displayed, and the resultant value passed to the assembler will be zero.

## 2.3 Accuracy

All expressions are evaluated to the accuracy of the BASIC language, that is, 13 significant digits. However, the result returned back to the Assembler, must always be a number between 0 to 65535. Expressions evaluated outside this range, at completion of the RPN process, will result in an error message to be displayed. Positive numbers within this range are always truncated to an integer value.

```
          8/(2*5) = 0          ( Integer values on return)
          8-10    = Error      ( Negative number)
          8*.25   = 2          ( Fractions permitted within expression)
TARGET    EQU   8*.25          ( TARGET now assumes the value 2)
SENSE     EQU   TARGET*TARGET  ( SENSE is set to the value 4)
```

## 2.4 Symbology

The Assembler is quite flexible in the handling of symbols. A symbol may start with any non-numeric character, other than the ones defined in sections 2.2, and be up to 16 characters in length. A symbol may be equated to a value in one of two ways, Inferred or Set.

The inferred method occurs automatically through processing a command line.

```
          ORG  $4000

START     JSR     SELECTCRT
          JMP     MSG:1
STOP      SET     R0 < 10
```

The symbol START would be assigned the "address" $4000. Note thatin this case, we do not use the reference 'value', because the symbol START actually refers to an address in Control Memory. The symbol STOP would now be referenced as $4002.

The set method is performed by executing the pseudo opcode EQU, or equate.

```
SELECTCRT    EQU    $802B
MSG:1        EQU    1024
```

The symbol SELECTCRT is assigned the VALUE $802B, while the symbol MSG:1 is assigned the VALUE 1024, or $0400. The assembler, when referencing the symbols, will use their set values.

Once a symbol has been evaluated, either through inference or by the EQU method, it cannot be redefined. Duplicate Symbol error messages will be displayed at the first attempt to redefine the symbol. Furthermore, usage of the symbol by any opcode will create the error message:

Attempt to use a Duplicate Symbol

Symbols, to be tested as symbols, must start in column one of the source line.

## 2.5 Syntax of Command line

All lines entered through the Assembler for processing are scanned in the following method:

```
(SYMBOL)      OPCODE   (Arguements)    (* comment field)
```

The scanning of the line first evaluates whether the whole line is a comment field. If the first character of the line is a * symbol, the entire line is treated as a comment field, and the line is printed, but not processed.

If the first character is not an * symbol, the computer determines if the character is a non-space code. If it is not a space code, all characters from position one in the line, to the first space code are treated as a symbol. The symbol is set into the Symbol table for processing later. A check for duplicate symbols only occurs during pass 1 of the assembler.

A scan is made for a trailing comment field, that is, text starting with the symbol ' * '. The space code prior to and after the * insures that the mathametical symbol times (*) is not taken as a comment. If a comment field is found, this is stripped from the line and placed into the comment buffer.

The command processor scans the line and tries to find an opcode. If an opcode is not found, the pseudo opcode processor attempts to find a match for the command. If it still fails, an error message is displayed.

### 2.5.1 Syntax of JMP,JSR commands

The JMP,JSR and BRA commands have the simplest structure:

```
JMP
JSR  } ( Address )
BRA
```

Where the Address field may be a Symbol, or numerical expression.

```
JMP    $5000       * Jmp to location $5000
JSR    TARGET      * JSR to location defined by TARGET
BRA    *-5         * Branches backwards to current location
                     -5
BRA    TS+.+10       * JMP to value of TS symbol plus
                       current location plus 10
```

### 2.5.2 Syntax of RTS instruction

The RTS instruction has a complicated structure, or a very simple structure.

```
      (,RC) (,RD)
RTS   (,WC) (,W1,BREG)
            (,W2,BREG)
```

Where ,RC Read Control Memory
,WC Write Control Memory
,RD Read Data Memory

,W1,BREG Write one byte a current PHPL
,W2,BREG Write one byte at (PHPL XOR 1)
Where BREG is R0,R1,R2,R3,R4,R5,R6,R7,PH,PL,CH,CL,SL,SH,K OR 0

```
RTS                 * Simple return
RTS   ,RC           * Read Control Memory, Return
RTS   ,WC           * Write Control Memory Return
RTS   ,RD           * Return, Read Data Memory
RTS   ,W1,R0
RTS   ,W2,K
RTS   ,RC ,W1,PL
```

### 2.5.3 Syntax of TPA,TPS,XPA instructions

This group allows the transfer of the PHPL pair to either the stack or an Auxillary register.

```
TPA    AR XX   (,RD)   (,+1) (,+2) (,+3) (,-1) (,-2) (,-3)
XPA            (,W1,BREG)
               (,W2,BREG)


TPS     (,+1) (,+2) (,+3) (,-1) (,-2) (,-3)
        (,RD)
        (,W1,BREG)
        (,W2,BREG)


XPA    AR 00  ,RD
TPA    AR 1D  ,W1,R5
TPA    AR 0E  ,+3 ,RD

TPS    ,+1
TPS    ,-3  ,RD
```

### 2.5.4 Syntax of Branch instructions BTL,BTH,BFH,BFL,BEL,BEH,BNL,BNH

```
OPCODE     (Value),BREG   (Address)
```

Where Value is in the range of 0 to 15, and the Address equates to a value within the current map segment.

### 2.5.5 Syntax of Branch Instructions BNR,BER,BLR,BLER,BLRX,BLEX

```
OPCODE    (AREG),(BREG)    (Address)
```

Where AREG is one of the valid A register gating options:

```
R0 - R7,CL-,CH-,CL,CH,CL+,CH+,00+,00-
```

Where BREG is one of the valid B register gating options:

```
R0-R7,PL,PH,CL,CH,SL,SH,K,0
```

Where Address is a value within the current map.

```
BER   00+,K    TARGET
BNR   R0,R1    .+15
BLER  R3,CH    .-5
BLRX  R4,R6    $4002
```

### 2.5.6 Syntax of TAP,TSP instructions

```
TAP   AR xx  (,RD)
             (,W1,BREG)
             (,W2,BREG)

TSP   (,RD)
      (,W1,BREG)
      (,W2,BREG)

TAP  AR 1F
TSP  ,RD
TAP  AR 03 ,W1,00
```

### 2.5.7 Syntax of Immediate Instructions

Opcodes IOR, IADD, IXOR, IAND, IADC, IDSC, IDAC, SET

```
OPCODE    (DREG) < (VALUE),(BREG)    (,RD)
                 =>                  (,W1)
                                     (,W2)
```

Where DREG is R0-R7,PL,PH,DUM,SL,SH or K
Where VALUE is between 0 and 255

```
IXOR    R0 < $0F,R1
IADD    R1 < $FF,R1   ,W1    * R1 = R1-1, WRITE RESULT TO DM
SET     R0 < $33             * Note special case
SET     R4 < RETURN
IDAC    K  < 4,PL     ,RD
```

### 2.5.8 Syntax of Register Instructions

Opcodes OR, AND, XOR, SBC, ADC, DAC, DSC and SDC
ORX, ANDX, XORX, SBCX, ADCX, DACX, DSCX SDCX

```
OPCODE     (DREG) < (AREG),(BREG)  (,CC) (,RD)
                                   (,CS) (,W1)
                                         (,W2)
```

```
ANDX    R6 < 00+,PL
XORX    PL < R0,R2
ORX     R5 < K,00    ,W1
ADC     R0 < R0,R1   ,CC
SBCX    PL < CL-,PL  ,CS
```

### 2.5.9 Syntax of SDC,MUL instructions

```
SDC    (ALBL)    (DREG) < (AREG),(BREG)   (,RD)
SDCX   (ALBH)                             (,W1)
MUL    (AHBL)                             (,W2)
MULX   (AHBH)
```

```
SDCX       R0 < R0,R1          * Note ALBL assumed
MUL        R2 < CH-,K   ,W1
SDC AHBH   PL < R5,CL
MUL ALBH   SH < SH,SL   ,RD
```

### 2.5.10 Syntax of IMUL instructions

```
IMUL   (ALBL)   (DREG) < (VALUE),(BREG)   (,RD)
       (ALBH)                             (,W1)
       (AHBL)                             (,W2)
       (AHBH)
```

Where VALUE is between 0 and 15

```
IMUL   ALBH   R4 < 4,PL
IMUL   ALBL   PL < 7,K   ,RD
```

NOTE: The operators AHBL and AHBH are really invalid in the context of the IMUL. They will be treated as an invalid datum and an error message will be reported.

### 2.5.11 Syntax of LPI instruction

```
LPI     (VALUE)  (,RD)
                 (,W1)
                 (,W2)
```

```
LPI     TARGET ,RD
LPI     100 ,W1
LPI     $8000  ,W2
LPI     300
```

### 2.5.12 Syntax of CIO instructions

```
CIO     (OBS)    (,RD)
        (CBS)    (,W1,BREG)
        (ABS)    (,W2,BREG)
        (CAB)
        (TIM)
```

```
CIO     CAB,ABS
CIO     TIM
CIO     OBS  ,RD
```

## 3.0 Control Memory Pseudo Op Codes

Several Pseudo Op Codes are available to the user to modify or alter the assembly of the source document. The following is a list of current codes:

| | |
|---|---|
| ORG | (Origin Control Memory) |
| LST | (Enable Listing Option) |
| NLST | (Disable Listing Option) |
| COPY | (Invoke Macro File Processing) |
| LOAD | (Chain to another Source file) |
| FCB | (Fix Control Bytes) |
| EPAR | (Enable Odd parity generation) |
| WPAR | (Enable Even parity generation) |
| NPAR | (Disable Parity Generation) |
| END | (Terminate processing of Source code) |
| EQU | (Equate symbol to expression) |
| STIT | (Enter Subtitle) |
| PAGE | (Eject current page and insert new Title) |
| FCW | (Fix Control Words) |
| RCW | (Reserve Control Memory Locations) |
| BLK | (Form exact 4096 load block) |
| DISP | (Write CRT display Record for Object file) |
| TAB | (Set TAB stops for listing) |

## 3.1 ORG (Origin)

The Current location in Control memory to assemble code is modified by this code. The expression to the right of the ORG statement is evaluated, and the current location counter is set to the value. Forward referenced symbols are not permitted, as the location of instructions ahead of the ORG statement have not been processed as yet.

```
                ORG   $1000          * Resets to 1000

1000  800000       FCB   $0

                ORG   256

0100  000001       FCB    1


                ORG   (.+1024)&$FC00     * Sets current location counter
                                         * to next "map".

0400  000002       FCB    2
```

### 3.2 LST (LiSTing Enable)

The LST Pseudo Op code is used to enable the listing option during Pass 2 of the assembler. LST is the normal default for the assembler. However, it may be used with the body of the text to enable the listing option after the NLST (No List) command was invoked.

### 3.3 NLST (No LiSTing)

NLST turns of the listing feature during pass 2 of the Assembler. It is normally used to prevent listing routines which have been tried, and would be extraneous to the content of the listing. NLST suppresses the incrementing of the line counter for listing control. NLST does not stop the processing of code within the NLST area though. Printing continues during NLST to device 000.

E.G.

```
2000  87800F    RTS

                NLST

 (Listing is suppressed here)

                LST

2100  DC0060    JMP $6000
```

### 3.4 COPY (Filename)

Macro calling of routines is enabled by the use of the COPY command. When discovered by the Assembler, current text position and file data are pushed onto an internal stack. The Filename specified in the arguement is opened, and code evaluation begins with the first line of the new file. When an END statement is encountered within the "COPY" file, the internal stack is popped, and evaluation of code commences at the correct position in the calling file.

The COPY file may itself call another COPY file. The stack is capable of handling 6 nested COPY commands before displaying an Error condition. The purpose of these so called Macro commands is to permit the building of library modules that perform standard functions. These modules can then be assembled with other modules/source code to permit powerful processing capabilities.

The user must be reminded that they alone must ensure Map boundaries are not overflowed for branches within the COPY file. It is advised that once a routine has been written and tested, the first series of instructions will origin the file at the next map location.

Furthermore, since a library routine, once written and tested, generally will not change, except for where it resides in memory, that the user should include NLST,LST commands to prevent re-listing these on the listing pass.

### 3.5 FCB (Fix Control Bytes)

FCB permits the user to enter in hex data to control memory locations for use as data. When used, parity is maintained and inserted into the high order bit if parity is enabled. Data values must be in the range of 0 to 65535. Error messages will result if data is outside this range.

```
          ORG  $4000

TARGET    EQU   $40

4000  001113        FCB    $1113
4001  804001        FCB    .                 * SET CONSTANT
4002  000040        FCB    TARGET            * EQUATE
```

### 3.6 EPAR (Enable PARity)

Used in the Control Memory section of the assembly only and enables the generation of the correct (ODD) parity bit for Control Memory words. EPAR is the default option for the Assembler.

### 3.7 WPAR (Wrong PARity)

WPAR will generate the wrong (EVEN) parity for Control Memory words. This is valid only when producing diagnostics in the Prom area, as the bootstrap proms will not load any file containing words of wrong parity. Furthermore, if used as an instruction, an automatic vector to the proms would take place. However, this instruction was implemented to permit the assembly of Prom code, which requires location $800C to be wrong parity for Power up tests.

### 3.8 NPAR (No PARity)

A constant 0 is inserted into the parity bit of each instruction following this pseudo-op. Parity is not checked and may be either right or wrong depending upon the bit pattern. NPAR is left here as a diagnostic tool.

### 3.9 END

This pseudo-op code is required at the end of each source file and will terminate processing of that file. It is imperative to state that you must insert this code on any file that is used as a Macro file under the COPY command. If using the EDITOR program supplied by Computer Concepts Corporation, the END statement may be left out. The Assembler will automatically detect the end of code and supply the termination logic.

## 3.10 EQU (EQUate)

Assignation of constant data to a symbol is the function of EQU. Often, external references to subroutines, error handlers, or other code must be supplied to the Assembler to process data. By using the EQU command, this data forms a record of the external references or provides more meaning than numbers would. It is not required to place EQU commands in the front of files. They may be used and interspersed anywhere within the source document. However, no forward references are allowed.

```
TARGET    EQU    $40          * Assigns the value $40
ZERO      EQU    0
ONE       EQU    1
LOCATE    EQU    .            * Sets LOCATE to current CM/DM location
ABC       EQU    LOCATE+TARGET
YES       EQU    YESTERDAY    * Illegal - YESTERDAY has not been
                              * Defined as yet


          SET    DUM < ZERO   * Much more meaning
          LPI    LOCATE       * Brings base address to PHPL
          SET    R0 < ONE
```

## 3.11 STIT (Ascii String)

STIT permits the user to insert Sub-titles into the listing. The Ascii string following the pseudo-op is inserted into the Sub title field. The next page eject will cause the Heading of the listing to include the new Sub Title. Note that any PAGE command will clear out the Sub Title field.

STIT Routine to handle the generation of PI

## 3.12 PAGE (Ascii String)

The PAGE command has several functions. First, the current listing page is terminated and a Top of Form command is given. Second, all current PAGE and STIT fields are cleared. The new ASCII string, if supplied, is inserted as the Main heading, and the new heading is displayed.

PAGE commands, which only affect listings, serve to segment logical areas of program code.

### 3.13 FCW (Fix Control Words)

FCW differs from FCB by not limiting the user to values between 0 to 65535. FCW permits the assigning of values to the full 23 bits of available Control Memory. Furthermore, Text strings may be entered into Control Memory packed three bytes to one CM word.

If a text string is greater than three bytes in length, multiple CM words will be generated to handle the overflow. Furthermore, if an even multiple of three bytes is not adhered to, the remaining bytes will be zeroes. This feature may allow the user to flag the end of a text string by reading a zero byte.

If a text string is an even multiple of three bytes in length, and extra Control Memory word of all zeroes will be appended to the string.

```
6300  123456      FCW   $123456
6301  800000      FCW   $7654321      * An error, exceeds three bytes
6302  C14200      FCW   'AB'          * Note the zero fill
6303  414243      FCW   'ABC'
6304  800000                          * Added because multiple 3
6305  414243      FCW   'ABCDE'
6306  C54600                          * Multiple lines handled
```

### 3.14 RCW (Reserve Control Words)

RCW allows the user to insert, or reserve, n number of zeroes into control memory.

```
          ORG   $4200

4200  800000        RCW   512
4400  000001        FCB   1
```

Though not displayed, all locations between 4200 and 43FF will contain the value $800000.

### 3.15 LOAD (Chain (LOAD) source code file)

Assembly of the current Source file is terminated, and the file defined by the LOAD command is opened and assembly commences at line one of this file.

```
LOAD   SEGM.01
```

### 3.16 BLK (Form modulus 4096 load block)

The BLK instruction should be little used. The purpose is to form a trailing record(s) of zeroes in a control memory load object file. When seen by the assembler, a flag is set. At the end of the Pass 2 assembly, if the flag is set, and an output object file is enabled, the program will output $800000 codes till the next modulo address of 4096 occurs. This allows us to load the object file, replete with checksums locations to the PATCHER program for processing.

However, if the user has elected to do scatter loading, the BLK command would cause only excess code to be appended to the end of the object module.

### 3.17 DISP (Display Record)

The DISP record dumps text following the DISP command to the disk in a format that allows the bootstrap proms to display to the CRT. This is similar to the message you see when the @@ file is loaded in. This is enabled only at the beginning of the file. The first time you process and code, either Data Memory or Control Memory, the execution of this opcode will create an error.

```
DISP  Loading User Diagnostic for Widgets
```

### 3.18 TAB (Set Tab stops for listing)

The TAB command allows the user to set new TAB stops for the listing. A series of arguements is given, and new TAB stops are entered.

The default Tab stops are as follows:

```
10,20,30,40,50,60,70,80,90,100,120,130,140,150,160
```

If the Tab stop given was greater than the following Tab stops, the following Tab stops are incremented by 1. If Two commas are encountered in a row, that Tab stop is not altered.

```
TAB    10,25,30,,55
```

Modifies 1st tab to 10, 2nd to 25, 3rd to 30, does not touch the 40 tab stop, but changes the 5th stop to 55. The remaining tab stops are not altered unless the last tab stop is greater than the next higher one.

## 4.0 Data Memory Pseudo - Op Codes

Assembler allows certain Pseudo Op codes for use in the Data Memory area. These codes are as follows:

| | |
|---|---|
| DORG | (Origin in Data Memory) |
| FSB | (Fix Single Byte) |
| FDB | (Fix Double Byte) |
| RMB | (Reserve Bytes) |
| FST | (Fix String) |

## 4.1 DORG (Data Memory Origin)

DORG permits the user to set a new origin in Data Memory for assembly of code. DORG sets flags which will create an error if any Non Data memory op code, or pseudo op code is executed.

```
          DORG  $1000          * Sets location counter to $1000

          DORG  COMPUTER       * The symbol COMPUTER is used to
                               * set the location counter in DM.
                               * COMPUTER may not be a forward
                               * reference.

          DORG  (.*20)+3       * Math is permitted
```

## 4.2 FSB (Fix Single Byte)

FSB will set the current Data Memory location to the value of the expression to the right. The expression must evaluate to a value between 0 to 255. Values outside this range will cause an error message to be printed.

```
          DORG    $C000

C000  32        FSB   '2'     * Text character definition
C001  0D        FSB   $D
C002  0F        FSB    15
C003  C0        FSB    ./256

C004  41        FSB    SYMBOL  * Where SYMBOL was equated as
                                * being 41 hex.

C005  00        FSB    $301    * Illegal, Value > 255
```

## 4.3 FDB (Fix Double Byte)

Similar to FSB, FDB permits two bytes to be assigned. The significance of FDB allows full 16 bit binary numbers to be stored in Data memory as pointers, or data.

```
                DORG   $D000

D000  0D0A    MSG:1       FDB    $0D0A
D002  4142                FDB    'AB'
D004  6364                FDB    'cd'
D006  FFFF                FDB    65535
D008  D000                FDB     MSG:1
```

## 4.4 FST (Fix String)

A string of text may be entered into Data memory by using the FST pseudo op code. The string must start and end with the ' symbol. When displayed, only three bytes of the text string will be displayed.

```
                DORG   $3100

3100  414243        FST  'ABCDEFGHIJK'
310B  00            FSB  0
```

## 4.5 RMB (Reserve Bytes)

Often, it is required to reserve an area of data memory for use as an array, buffer area, or temporary storage area. RMB allows any number of Data memory locations to be allocated for this purpose.

```
                DORG   $4350

4350  0100    BUFFER        RMB    256
4450  AA                    FSB    $AA
```

No data is actually inserted in the reserved area.

## 5.0 Conditional Assembly Codes

Assembler supports several conditional assembly directives, enabling the programmer to minimize source file segmentation. The following is a list of the Conditional Assembly Codes:

| | |
|---|---|
| IFDF | (IF Symbol has been previously defined) |
| IFND | (IF Symbol has not been previously defined) |
| IF | (IF (Value) (Condition) (Value) |
| ELSE | (Opposite Condition) |
| FIN | (End conditional Assembly) |

Conditional assembly codes may not be "nested". That is, once invoked, another IFDF,IFND or IF statement may not reside within the conditional block.

### 5.1 IFDF (SYMBOL)

Conditional Assembly of code following the IFDF (Symbol) code will take place ONLY if the symbol following the IFDF code has been previously defined.

This allows the programmer to construct BASIC equivalent statements such as IF ... THEN .... ELSE. If the symbol following the IFDF statement has not been defined PREVIOUS to this encounter, no code is assembled till the Conditional code FIN is encountered. The only exception to this is the ELSE statement.

A symbol defined AFTER the occurance of the IFDF statement has no effect on the conditional logic. However, if after the IFDF statement has failed, and the symbol has been defined, through inference or by Equating, AND another Conditional test statement of any kind (IFDF,IFND or IF) uses that symbol, then invalid object code will be produced. The Conditional codes are meant to test a symbol once, and once tested, generally are not tested again.

The symbol does not have to be defined by the current segment. A prior segment may have defined the symbol. IFDF, and also IFND, do not care what the value of the symbol is, only that the symbol has been defined.

As an example, the following page has a program that will assemble code depending upon whether the system is meant for an LVP or SVP system. The Main module will call the correct routine.

Example of IFDF condition coding:

```
(Main Routine)
          COPY   LVP
          COPY   TARGET

          END

(LVP Routine)

  0001    LVP     EQU    1          * Define LVP

                END

(TARGET Routine)

                IFDF   LVP

  0002    DISK   EQU    2

                ELSE

   *S     DISK   EQU    1

                FIN               * END CONDITIONAL ASSEMBLY
```

Note that if the symbol LVP had not been defined, the symbol DISK would have been evaluated to 1, not 2. The *S symbol implies that the code is being Skipped over.

## 5.2 IFND (SYMBOL)

IFND reacts much like the IFDF statement, except it has the opposite meaning. That is, the symbol must not have been defined prior to using this Conditional Code. If the symbol had been defined, no assembly would take place till the Code ELSE or FIN were encountered.

The IFND code is especially useful when creating default symbols in a segment. If the symbols were not defined in previous sections, the IFND would create them.

```
                IFND    LINEPRINTER

0204    LINEPRINTER     EQU   $204

                FIN
```

### 5.3 ELSE statement

ELSE switches the conditional logic to the opposite state when encountered. That is, if assembly was turned off due to a previous conditional statement, the assembly is turned on. If the assembly was on, ELSE turns it off. ELSE must be executed within a Conditional assembly, prior to encountering FIN.

It should be noted that it is not REQUIRED to insert a ELSE statement in every conditional. Examples in section 5.1 and 5.2 show usage of conditional statements with and without the ELSE statement. ELSE may be used with the IFDF,IFND and IF statements.

### 5.4 FIN (End Conditional Assembly)

FIN terminates the current block of Conditional Coding. The occurance of another FIN or ELSE statement will result in an error statement. ALL conditional assembly must terminate with the FIN statement.

### 5.5 IF (Value) (Condition) (Value)

The IF statement is a most powerful conditional code statement. The basis of assembling the code following the statement is dependant upon the mathametical evaluation of the two values following the IF statement. Each value may be a singular or group of PREVIOUSLY DEFINED Symbology, Numeric, Current Location pointers, Literal or Hex data. The first value expression must preceed the Condition arguements by at least one space. Failure to do so will result in an invalid arguement error code. The evaluations permitted are:

| | |
|---|---|
| = | Equality |
| > | Greater than |
| < | Less than |
| >= | Greater than or Equal to |
| <= | Less than or Equal to |
| <> | Not Equal |

```
Example         IF   LINEPRINTER = $204
```

The Symbol LINEPRINTER, which was previously defined, will be looked up. The value of the symbol is then compared with the Hex value $0204. If the values are equal, the code following will be assembled. If the values are not equal, no code is assembled until the operators FIN or ELSE are encountered.

Again, the usage of symbology in an IF statement implies that a PREVIOUSLY defined Symbol will be used.

## 6.0 Symbolic Dumps

At the end of any assembly, successful or not, that is printed on console or Lineprinter devices, a complete symbol dump, both in Alphabetical order, then in Numerical order will be printed.

This symbol dump is a handy reference tool for use in debugging code generated. The format for symbol dumps is as follows:

```
$1234  START:ADDRESS
  !   !     !--------------  Symbol Name
  !   !-------------------- Error Code
  !------------------------ Address or Value Assigned to Symbol
```

The error codes assigned at present are:

| | | |
|---|---|---|
| " " | (Space) | Normal Assignement |
| * | (Asterisk) | Valid but not used by current Program |
| # | (Pound) | Duplicate Symbol |

As noted, the first two are not in error. The * code simply means that no references to that symbol were made during the course of an assembly. The pound (#) symbol does imply an error condition. Two seperate equates, by inference or Equating, or combinations therein were made to that symbol. The assembler will only display the first value assigned.

Symbols are stored outside of the catalogued area on the disk, using a hash algorithm. There is no practical limit to the number of symbols that may be stored. (Limited only to the size of the area after the catalogue, and your patience.) However, more than 2000 symbols will slow the compilation significantly.

" EXTERNALLY DEFINED
@ Public Symbol DEFINED
! UNDEFINED IN CURRENT ASSEMBLY