# FUTUREDATA RDOS V1.2

Addendum

FUTUREDATA COMPUTER CORP.
11205 S. La Cienega Blvd.
Los Angeles, CA 90045

# RDOS VERSION 1.2 REVISION NOTICE

RDOS Version 1.2 includes the following enhancements:

A. Macro facility

B. Assembler displays English language error messages

RDOS Version 1.2 contains the following corrections:

A. Z-80 Assembler properly assembles indexed instructions.

B. Linkage Editor runs properly with a command file.

RDOS Version 1.2 is the generic name for a set of programs. Each individual program may have a different version number.

**PLEASE DESTROY ALL PREVIOUS COPIES OF RDOS INCLUDING ANY PRELIMINARY COPIES OF V1.2.**

08/28/78

FUTUREDATA RDOS

The Futuredata RDOS consists of three new programs, ASMR, a relocating macro assembler, LINK, a linkage editor, and DEBUGR, a debugger which supports In Circuit Emulation and Symbolic Debugging.

## 1. RDOS MACRO ASSEMBLER

Run the Assembler by typing JA. The available options will be displayed on the screen.

### 1.0 ASSEMBLER OPTIONS

L - List the Assembler output on the CRT.

M - Use a macro library file.

O - Write a relocatable file on disk.

T - Produce a truncated listing on the CRT. Lines will be limited to 40 characters and DC statements will print only one line.

E - List and/or print only the lines which have an error flag.

S - Include the symbol table at the end of the relocatable object file.

Z - This option will flag all lines with non-8080 or 8085 opcodes.

P - Print the listing on the Microprinter.

B - Send the listing to the Serial Port.

## 1.1 MACROS

The macro facility allows a user to define an opcode which actually causes a series of instructions to be assembled. For example, using the 8080, the following series of instructions will add 10 to the H and L registers, using only the A register.

```
                MOV     A,L
                ADI     10
                MOV     L,A
                MOV     A,H
                ACI     0
                MOV     H,A
```

If the user finds many such sequences in his programs, he will find it more efficient (of his time and disk space) and less error prone to define a macro that will cause the sequence of instructions to be assembled whenever its name appears in the opcode field:

```
ADD10           MACRO
                MOV     A,L
                ADI     10
                MOV     L,A
                MOV     A,H
                ACI     0
                MOV     H,A
                ENDM
```

The macro definition is begun by the MACRO pseudo-op. The label field of the MACRO pseudo-op defines the name of the macro, in example 1, ADD10. Statements following the MACRO pseudo-op represent the body of the macro definition. These statements will be processed by the assembler when the macro name occurs in the opcode field (when the macro is 'called'). The ENDM pseudo-op ends the macro definition. The macro definition must be placed at the beginning of the assembler source file, only the EJE, SPC, and PRNT pseudo-ops may occur before the macro definitions.

### 1.1.1 MACRO PARAMETERS

Obviously, a macro definition like ADD10 is of limited use. How many times would you add the number 10 to the H and L registers? It is much more likely that similar rather than identical sequences of instructions occur in a program. Allowing a macro definition to have variables that may have various values when the macro is called is one way of doing this. For example, if we make the actual value added to the H and L registers a macro parameter, the user can define a macro that, when called, will generate statements that will add any number from 0 to 255 to the H and L registers:

```
ADDX                    MACRO     NUM
                        MOV       A,L
                        ADI       &NUM
                        MOV       L,A
                        MOV       A,H
                        ACI       0
                        MOV       H,A
                        ENDM
```

If the macro is called in the following way:

```
                        LHLD      VALADR
                        ADDX      5
                        SHLD      VALADR
```

The following statements will be generated:

```
                        LHLD      VALADR
                        MOV       A,L
                        ADI       5
                        MOV       L,A
                        ACI       0
                        MOV       H,A
                        SHLD      VALADR
```

What happens is that in the macro definition, the names of all the macro parameters are listed (separated by commas) in the operand field of the MACRO pseudo-op. When the macro is called, the actual values to be used are listed in the operand field of the calling statement (separated by commas). Then, wherever a macro parameter name appears in the body of the macro definition, preceeded by an ampersand, the actual value of the parameter is substituted in place of the ampersand and name. Note that this substitution is done character for character (up to 32 characters per parameter). The macro parameters need not be numeric. Also, the substitution may occur in the label, opcode, operand, comment, or any combination of fields. For example, for the macro definition:

```
SAVEA                   MACRO     TYPE,LOC
                        &TYPE     &LOC
                        ENDM
```

The call:

```
                    SAVEA    STA,VAL
```

will result in:

```
                    STA      VAL
```

while the call:

```
                    SAVEA    STAX,D
```

will result in:

```
                    STAX     D
```

1.1.1.1 MACRO PARAMETER DELIMETERS

As stated, when a macro is called, the actual values of the parameters are listed in the operand field of the calling statement, separated by commas. However, commas and blanks may exist in a parameter if they occur between quotes. Also, there must be an even number of quotes in any parameter. The following are examples of valid parameters:

```
                    20
                    X'40'
                    'A,B'+'B,C'
                    A''C
                    ABC
```

1.1.2 GENERATING UNIQUE LABELS

A macro call may generate statements with labels. For example, the following definition: (to make D,E=absolute value of D,E)

```
ABSD                MACRO
                    MOV      A,D
                    ORA      A
                    JP       ENDABSD
                    CMA
                    MOV      D,A
                    MOV      E,A
                    CMA
                    MOV      E,A
                    INX      D
ENDABSD             EQU      *
                    ENDM
```

The first time ABSD is called, label ENDABSD will be defined. Unfortunately, the second time ABSD is called, ENDABSD will also be defined and a duplicate definition error will result. What is needed is a way to generate a unique label each time the macro is called. The Futuredata Macro Assembler provides this capability by pre-defining a special parameter - INDX - which will always have a unique 5 digit numeric value each time a macro is called. The macro ABSD may now be defined in the following way:

```
ABSD                 MACRO
                     MOV       A,D
                     ORA       A
                     JP        AB&INDX
                     CMA
                     MOV       D,A
                     MOV       A,E
                     CMA
                     MOV       E,A
                     INX       D
AB&INDX              EQU       *
                     ENDM
```

Assuming that this is the only macro defined and that there are exactly two calls made, the first call defines the label AB00001 and the second call defines the label AB00002. Note that a label on the actual macro call statement takes on the current location counter value at the beginning of the call.

## 1.1.3 CONCATENATING PARAMETERS

In the previous example, the value of parameter INDX was appended to the right of the characters AB simply by writing the parameter name, preceeded by an ampersand, to the immediate right of the characters AB: AB&INDX. If, also, the user wished to append an A to the right of the value of parameter INDX, a problem would have arisen:

```
                     JP        AB&INDXA
```

The macro processor would have interpreted this to mean: Substitute the value for parameter INDXA. To solve this problem, the macro processor recognizes a parameter name delimiter: ! whose only function is to indicate the end of a parameter name. The correct way to append an A to the right of the value of parameter INDX is:

```
                     JP        AB&INDX!A
```

On the first call, this would result in the statement:

```
                    JP        AB00001A
```

Note that if ! appears anywhere other than after a macro parameter (or a set symbol), it will be treated as a normal character.

### 1.1.4 LITERAL AMPERSAND

Since the ampersand signals the macro processor to substitute a parameter value, a special set of characters is needed to tell the macro processor that rather than substituting, an actual ampersand character is wanted. For example, if the user wants to put an ampersand into the A register, he might think to write:

```
                    MVI       A,'&'
```

However, for reasons explained under Advanced Considerations, whenever a single ampersand is wanted in the generated statement, four ampersands must be written in the actual source:

```
                    MVI       A,'&&&&'
```

### 1.1.5 MACRO LIBRARY

All macro definitions must be placed at the beginning of the source input file. However, by specifying the M option, a second source file of macro definitions may be included in the assembly. As in the main input file, the first statement other than EJE, SPC, and PRINT which is not in a macro definition will terminate reading of macro definitions.

### 1.1.6 DUPLICATE MACRO DEFINITIONS

If two macros with the same name are defined, the last macro definition read will be the one used. Thus macro definitions in the main source file take precedence over macro definitions in the macro library.

### 1.1.7 MACRO CALLS WITHIN MACROS

A macro definition may include a statement which calls another (or the same) macro. These calls may be nested to any level (depending on the symbol table space available). A macro definition may not have another macro defined in its body.

### 1.1.8 EXITM STATEMENT

When processed, the EXITM statement causes immediate termination of the current or named macro. The next statement processed will be the first one following the macro call. The syntax is:

```
                    EXITM [<macro-name>]
```

Note that no label is allowed.

## 1.2 CONDITIONAL ASSEMBLY

These statements allow a programmer to selectively assemble statements in a source file. For example, if a programmer did not know whether the program was going to be used with a tape or disk, both the calls to the tape subroutine and disk subroutine could be included in the source file, and conditional statements used to actually assemble the one needed:

```
DEVICE                  DEFG     'TAPE'
                        .
                        .
                        .
                        IF       '&DEVICE'='TAPE'
                        LHLD     TAPEFCB
                        CALL     TAPEIO
                        ELSEIF   '&DEVICE'='DISK'
                        LHLD     DISKFCB
                        CALL     DISKIO
                        ENDIF
```

More often, conditional statements will be used to implement more complex macros. For example, the ADDX macro defined in section 1.1.1 could be expanded to add a number between 0 and 255 to the BC, DE, or HL registers:

```
ADDXY                   MACRO    REG,NUM
                        IF       '&REG'='BC'
REGH                    DEFL     'B'
REGL                    DEFL     'C'
                        ELSEIF   '&REG'='DE'
REGH                    DEFL     'D'
REGL                    DEFL     'E'
                        ELSE
REGH                    DEFL     'H'
REGL                    DEFL     'L'
                        ENDIF
                        MOV      A,&REGL
                        ADI      &NUM
                        MOV      &REGL,A
                        MOV      A,&REGH
                        ACI      0
                        MOV      &REGH,A
```

```
                    ENDM
```

ADDXY could be expanded further to add a number between 0 and 65537 to the BC, DE, or HL registers:

```
ADDXY               MACRO     REG,NUM
                    IF        '&REG'='BC'
REGH                DEFL      'B'
REGL                DEFL      'C'
                    ELSEIF    '&REG'='DE'
REGH                DEFL      'D'
REGL                DEFL      'E'
                    ELSE
REGH                DEFL      'H'
REGL                DEFL      'L'
                    ENDIF
NUMH                DEFL      &NUM/256
NUML                DEFL      &NUM.MOD.256
                    MOV       A,&REGL
                    ADI       &NUML
                    MOV       &REGL,A
                    MOV       A,&REGH
                    ACI       &NUMH
                    MOV       &REGH,A
                    ENDM
```

One of the more common uses of conditional statements is to generate tables that would be long, tedious, and error prone to enter by hand. The following example generates a table that could be used to check whether a character is numeric:

```
TABLE               EQU       *
CHAR                DEFG      0
                    DO        256
                    IF        &CHAR <'0'.OR.&CHAR>'9'
                    DC        0
                    ELSE
                    DC        X'FF'
                    ENDIF
&CHAR               DEFG      &CHAR+1
                    ENDDO
```

## 1.2.1 SET SYMBOLS

Set symbols (and macro parameters) are the variables used by the conditional statements. Their value may be either numeric (0 to 65537) or a character string (0 to 32 characters long). The values are set by using the DEFG (define global) and DEFL (define local) statements.

### 1.2.1.1 DEFL STATEMENT

This statement is used to define (and redefine) a local set symbol. A local set symbol is only known in the macro in which it is defined. A set symbol of the same name defined in a different macro is actually a different set symbol. The syntax is:

```
<set-symbol-name>   DEFL     <expression> ! '<string>'
```

If quotes are placed around the operand, it is treated as a character string and may be a maximum of 32 characters long. Quotes embedded in the string must be doubled. They are, however, stored as double, not single, quotes. If quotes are not placed around the operand, it is treated as an integer expression. Labels may appear in the expression only if they have been previously defined.

Note that when defining a set symbol, an ampersand must not preceed the name in the label field. When using the set symbol, an ampersand must preceed the name.

### 1.2.1.2 DEFG STATEMENT

This statement is used to define (and redefine) a global set symbol. A global set symbol is known in all macros unless a local set symbol of the same name is defined in a particular macro. In this case, the global set symbol will become unknown, in that particular macro only, as soon as the like named local set symbol is defined. The syntax is:

```
<set-symbol-name>   DEFG     <expression> ! '<string>'
```

If quotes are placed around the operand, it is treated as a character string and may be a maximum of 32 characters long. Quotes embedded in the string must be doubled. If quotes are not placed around the operand, it is treated as an integer expression. Labels may appear in the expression only if they have been previously defined.

Note that when defining a set symbol, an ampersand must not preceed the name in the label field. When using the set symbol, an ampersand must preceed the name.

### 1.2.2 IF BLOCK

An IF block begins with an IF statement and ends with an ENDIF statement. IF blocks may be nested within IF blocks or DO blocks to any level.

### 1.2.2.1 IF STATEMENT

The IF statement begins an IF block. It may have an optional name in the label field which may be referred to in the EXITIF statement. If the expression in the operand field of the IF statement is non-zero, the statements following the IF statement up to the first ELSEIF, ELSE, or EXITIF statement at the same nesting level will be processed. If the expression is zero, the statements following the IF up to the first ELSEIF, ELSE, or ENDIF statement at the same nesting level will be ignored. The syntax is:

```
[<if-block-name>]    IF        <expression>
```

### 1.2.2.2 ELSEIF STATEMENT

The ELSEIF statement is used in conjunction with an IF statement to test an alternate condition without going to a deeper nesting level. If the expression in the IF statement was 0 and the expressions in all previous ELSEIF statements at this nesting level were 0, and the expression in this ELSEIF statement is non-zero, statements up to the next ELSEIF, ELSE, or ENDIF statement at this nesting level will be processed. The syntax is:

```
                    ELSEIF    <expression>
```

Note that no label is allowed.

### 1.2.2.3 ELSE STATEMENT

The ELSE statement is used in conjunction with an IF statement to indicate the last alternative. It is identical to:

```
                    ELSEIF    1
```

That is, if the expressions in the IF statement and all subsequent ELSEIF statements at this nesting level are 0, the statements after the ELSE statement up to the closing ENDIF statement are processed. The syntax is:

```
                    ELSE
```

Note that no label or operand is allowed.

### 1.2.2.4 EXITIF STATEMENT

The EXITIF statement when processed will cause all statements up to the closing ENDIF statement in the named or current IF block to be ignored. The syntax is:

```
                    EXITIF   [<if-block-name>]
```

Note that no label is allowed.

#### 1.2.2.5 ENDIF STATEMENT

The ENDIF statement terminates an IF block. The syntax is:

```
                    ENDIF
```

Note that no label or operand is allowed.

### 1.2.3 DO BLOCK

A DO block begins with a DO statement and ends with an ENDDO statement. It causes repetitive assembly of the statements within the block. DO blocks may be nested within IF blocks or DO blocks to any level.

#### 1.2.3.1 DO STATEMENT

The DO statement begins a DO block. It may have an optional name in the label field which may be referred to in the EXITDO and NEXTDO statements. The expression in the operand field is evaluated ONCE at the entry to the DO block and is stored as the DO COUNT - the number of times the statements within the block are processed. The syntax is:

```
[<do-block-name>]   DO       [<expression>]
```

If the expression is omitted, the block will be processed 65538 times (essentially indefinitely).

#### 1.2.3.2 EXITDO STATEMENT

The EXITDO statements, when processed, causes the assembler to immediately terminate processing statements in the current or named DO block and begin at the first statement after the closing ENDDO statement. The syntax is:

```
                    EXITDO   [<do-block-name>]
```

Note that no label is allowed.

#### 1.2.3.3 NEXTDO STATEMENT

The NEXTDO statement, when processed, causes the assembler to immediately begin processing the next iteration of the current or named DO block. That is, if this is not the last iteration, the next statement processed will be the first statement after the DO statement. The syntax is:

```
NEXTDO   [<do-block-name>]
```

Note that no label is allowed.

### 1. 2. 3. 4 ENDDO STATEMENT

The ENDDO statement terminates a DO block. The syntax is:

```
ENDDO
```

Note that no label or operand is allowed.

### 1. 2. 3. 5 SUBSTR STATEMENT

he SUBSTR (substring) statement assigns a part of a string to a t symbol. If the set-symbol has not been previously defined, a ew local set-symbol will be defined. The syntax is:

```
<set-symbol-name>   SUBSTR   <expression1>,<expression2>,'<string>'
```

<expression1> defines the beginning character position of the substring. The first character is position 1. <expression2> defines the length of the substring. If <expression2> is 0, the substring will begin with the character defined by <expression1> and continue to the end of the string.

### 1. 2. 3. 6 LENGTH STATEMENT

The LENGTH statement assigns the length of a string to a set symbol. If the set-symbol has not been previously defined, a new local set-symbol will be defined. The syntax is:

```
<set-symbol-name>   LENGTH   '<string>'
```

## 1. 3 ADVANCED CONSIDERATIONS

Each source line is scanned twice before processing for ampersands which signal set symbol or macro parameter substitution. During ach scan any set symbol names preceeded by ampersands have their alues substituted for the ampersand and name. Any doubled ampersands are replaced by a single ampersand. This allows the following capabilities:

## 1.3.1 SUBSCRIPTED SET SYMBOLS

The following statement will define a set symbol whose name is based on the value of another set symbol:

```
A&I                     DEFL    'A'
```

If &I has the integer value 4, the set symbol A00004 will be defined to have the value A. The following statement uses the set symbol:

```
                        MVI     B,'&&A&I'
```

On the first scan, the && is replaced by & and &I is replaced by 00004 leaving:

```
                        MVI     B,'&A00004'
```

On the second scan, &A00004 is replaced by A leaving:

```
                        MVI     B,'A'
```

By varying the value of the set symbol I, set symbols may be defined and referenced which are indexed using I as a subscript.

## 1.3.2 INDIRECT SET SYMBOLS

The following statements will define a set symbol whose value is the name of another set symbol:

```
A                       DEFG    'ABC'
B                       DEFG    'A'
```

The following statement will use the set symbol B as an indirect reference to the value of set symbol A:

```
                        IF      '&&&B'='ABC'
```

On the first scan, the first two ampersands will be replaced by a single ampersand and &B will be replaced by A leaving:

```
                        IF      '&A='ABC'
```

On the second scan, &A will be replaced by ABC leaving:

```
                        IF      'ABC'='ABC'
```

## 1.4 ASSEMBLER PRINT STATEMENT

The PRINT statement controls which source lines are displayed and/or printed. There are now four possible operands.

### 1.4.1 PRINT OFF

Suppresses printing of all following source lines including itself.

### 1.4.2 PRINT ON

Causes printing of all source lines except for those in a macro or lines skipped due to conditional assembly pseudo-ops or the conditional pseudo-ops themselves: DEFG, DEFL, IF, ELSEIF, ELSE, ENDIF, EXITIF, DO, EXITDO, NEXTDO, ENDDO.

### 4.3 PRINT GEN

Causes printing of all source lines except for those skipped due to conditional assembly pseudo-ops or the conditional pseudo-ops themselves. A plus sign will be printed to the left of the location counter value for source lines contained in a macro.

### 1.4.4 PRINT ALL

Causes printing of all source lines including those skipped due to conditional assembly pseudo-ops and the pseudo-ops themselves. Lines skipped will have no location counter value or object code printed.

### 1.4.5 SUBSTITUTION

Lines are printed after macro parameter and set symbol substitution takes place.

## 1.5 ASSEMBLER EXPRESSIONS

Operands may now be general expressions containing the operators: +, -, unary -, *, /, (, ), .MOD., .SHR., .SHL., .AND., .OR., .XOR., and .NOT., as well as the relational operators =, >, <, >=, <=, ><. Character strings in quotes may be used as arguments with the relational operators. Blanks delimit the expression. Expressions are evaluated left to right. Up to 8 levels of parentheses are allowed. Operators with higher precedence are evaluated before operators of lower precedence that immediately preceed or follow them. The operator precedences are as follows:

```
7:Parenthesized expressions
6:*,/,.MOD.,.SHL.,.SHR.
5:+,-,unary -
4:=,>,<,>=,<=,><
3:.NOT.
2:.AND.
1:.OR.,.XOR.
```

Examples:

```
2+3*4 (result is 14)
(2+3)*4 (result is 20)
5.OR.X'A' (result is 15)
5.SHR.1 (result is 2)
.NOT.5*4 (result is X'EB')
'ABC'<'DEF' (result is 1)
'ABC'>'AB' (result is 1)
'ABC'='DE'.OR.'WXYZ'>47 (result is 1)
6300>'DEFG' (result is 0)
'A''B'='A''B' (result is 1)
```

Note that strings of 2 or less characters are treated as their numeric equivalents (i.e. 'AB' is equal to X'4142'). The null string ('') is equal to 0.

## 1.6 RELOCATION PSEUDO-INSTRUCTIONS

The RDOS Assembler contains the following additional pseudo-instructions to implement the relocation features:

ASEG

The ASEG statement is used to define an absolute segment. The default for the assembler is an absolute segment starting at location zero. After intervening RSEGs, an ASEG instruction will reset the instruction counter to the value of the end of the previous absolute segment.

RSEG <rseg name>

The RSEG statement defines a relocatable segment. Up to eight segments are allowed in one assembly, each identified by a name in the operand field. Each RSEG may be stopped and started again where they were left off by another RSEG statement with the same name.

GLBL <symbol>[,<symbol]...

The GLBL statement is used to list all the external references and entry points in an assembly. A maximum of 255 external references may be used in one assembly.

ORG <expression>

The ORG statement is used to reorigin the subsequent lines of code. The <expression> of the ORG statement must be absolute if it is in an absolute segment and relative to the start of the current RSEG if it is in a relocatable segment.

## 1.7 ASSEMBLER ERROR MESSAGES

English error messages are now displayed immediately below the line in error. More than one message may be printed for each line. They should be self explanatory.

## 2. RDOS LINKAGE EDITOR

Run the Linkage Editor by typing JL. The available options will be displayed on the screen.

### 2.1 LINKAGE EDITOR OPTIONS

| | |
|---|---|
| C | Input to the Linkage Editor is in a command file created using the Editor. The file contains the relocatable input filenames, the #ORG and #END statements and the RSEG's to be included. |
| O | An absolute object file is to be written on disk. |
| D | RSEG's that are not specifically listed are to be deleted from the output module. |
| S | The symbol table is to be read from each input file and then appended to the end of the absolute object file. The symbol table can then be loaded by the debugger and used for symbolic debugging. All relative addresses are relocated by adding the base address of the appropriate RSEG. |
| L | List the memory map and reference list on the CRT. |
| P | Print the memory map and reference list on the Microprinter. |
| B | Send the memory map and reference list to the Serial Port. |

If "C" is specified the Linkage Editor will prompt SPECIFY COMMAND FILE. Type the file name and RETURN.// If "C" was not specified the Linkage Editor will prompt SPECIFY INPUT FILE. Enter the filename and RETURN. The Linkage Editor will the open the file, list the RSEG's and their lengths on the screen and ask INCLUDE THIS FILE?. Type "Y" to include the file, "N" to ignore it. Once a file is included, you must not remove the disk containing the file. After processing your response, The Linkage Editor will again prompt SPECIFY INPUT FILE. If another file is to be included proceed as before, if not enter a RETURN.

If "O" is specified the Linkage Editor will prompt SPECIFY OBJECT FILE. Type the file name and RETURN.

The Linkage Editor will then prompt LINKER INPUT.

## 2.2 LINKAGE EDITOR INPUT

Enter the RSEGs and their locations in memory using the following format:

```
#ORG <absolute address>
<list of RSEGs to be included separated by commas>
#ORG <absolute address>
<list of RSEGs to be included separated by commas>
#END <entry point, must be a global symbol>
```

If "D" (delete RSEGs) is not specified all RSEGs that are not listed are included after the last #ORG statement.

If an entry point is not specified the default is the entry point of the first relocatable file to be included.

When the Linkage Editor is finished it displays the message FUNCTION COMPLETED.

## 2.3 LINKAGE EDITOR COMMAND FILE

The command file is created using the Editor. It supplies the Linkage Editor with the necessary information about input files and the desired location for RSEGs. The format of the command file is as follows:

```
<input filename>
<input filename>
.
.
<input filename>
#ORG <absolute address>
<rseg names separated by commas>
#ORG <absolute address>
<rseg names separated by commas>
.
.
#ORG <absolute address>
<rseg names separated by commas>
#END <entry point symbol>
```

## 2.4 LINKAGE EDITOR OUTPUT

If requested the Linkage Editor will output, to the CRT and/or the printer, a reference list and a memory map.

The reference list shows for each input file the RSEGs, their absolute addresses, and their lengths. It also lists all of the global symbols in each file and their absolute addresses. The format of the reference list is as follows:

```
                  FILE RSEG ADDR LENGTH



                  GLOBALS
```

The memory map shows the memory locations of the RSEGs. It lists them in the order in which they were entered in the linker input. So if the operands of the #ORG statements were in assending order the memory map will also be. The format of the memory map is as follows:

```
                  ADDRESS RSEG FILE LENGTH
```

If a RSEG was overlaid by another RSEG of the same name, it will be flagged by an "O" to the right of the length column. if there is more than one RSEG of the same name the first one encountered by the Linkage Editor will be used. If a RSEG has been deleted ("D" was specified and the RSEG was not listed) the entry in the memory map will be flagged with an "D".

At the end of the listing the entry point for the object module is given.

## 2.5 LINKAGE EDITOR ERROR MESSAGES

Error messages and their meanings are described in the following section.

NOT A SOURCE FILE - The command file is not an Editor source file.

NOT A RELOCATABLE FILE - The Linkage Editor input file was not created by the Futuredata Relocating Assembler or the "R" attribute has been changed using the Monitor.

PARM ERR...RESPECIFY - Syntax error in the linker input.

<label> **DUPLICATE GLOBAL IN <filename> - The first occurance of

a global label is used for address references. all additional definitions are flagged as an error.

<label> **UNRESOLVED REFERENCE IN <filename> - The external reference was not found in the global symbol table.

IN RSEG <rseg> **UNRESOLVED REFERENCE IN <filename> - Reference between RSEGs in the same assembly can not be correctly relocated. Probable cause is references to a RSEG that has been deleted.

**DELETED RSEG REFERENCED IN <filename> - An RSEG which was not included is needed to resolve address references.

**ERROR IN COMMAND FILE - Incorrect file name was specified or a syntax error in the command file.

**RELOCATION ERROR IN <filename> - Input file was not correctly assembled with the RDOS assembler.

**TABLE OVERFLOW - More memory is needed by the Linkage Editor.

**SYMBOL TABLE NOT FOUND IN <filename> - The symbol table was not included when the program was assembled.

**SEQUENCE ERROR IN <RELOCATABLE FILE NAME> - records in <relocatable file name> are not in proper order. Reassemble.

## 3. RDOS DEBUGGER

Run the Debugger by typing JD or JI. The screen will display memory centered at 0 in the same format as the Futuredata DOS Debugger. All commands are identical to those in the DOS Debugger with two exceptions necessary for symbolic debugging:

### 3.1 LOADING A SYMBOL TABLE

In order to have a symbol table included in an object file, you must select the option which causes it to be included both when Assembling and Linkage Editing. In order to load the symbol table you must specify its beginning address and whether you want all symbols or only global symbols at the time you load the program. The format of the load command is:

```
L <filename>[,[<offset>][,[<symbol table address>][,G]]]
```

entities enclosed in brackets are optional). If <offset> is not pecified, its default is 0. If <symbol table address> is not specified, but the preceeding comma is, its default is the currently displayed address. Examples:

| | |
|---|---|
| LTEST.L,,2000 | Loads symbols for TEST.L starting at X'2000' |
| L TEST.L,0,2000,G | Loads global symbols for TEST.L starting at X'2000' |
| L TEST.L,,,G | Loads global symbols for TEST.L starting at the currently displayed address |
| L TEST.L,100, | Loads symbols for TEST.L starting at the currently displayed address. Offsets program by X'100'. Does not offset addresses, however, and thus makes symbol table meaningless. |

After reading the symbol table, the Debugger prints the loaded length of the table at the bottom of the screen (in hex). Examples:

```
SYMTAB LENGTH=015A
GLOBAL SYMTAB LENGTH=004B
```

f you attempt to load the symbol table where there is no memory or where there is protected memory, the message 'PARTLY LOADED' will be appended to the length message. Example:

```
SYMTAB LENGTH=201A; PARTLY LOADED
```

In this case only part of the symbol table could be loaded. You may reference those symbols loaded. in order to load the complete symbol table, you must either add more memory or specify a lower symbol table address.

## 3.2 REFERENCING A SYMBOL

You may use a symbolic reference wherever you can use a hexadecimal address. The syntax of a symbolic reference is:

```
#[<relocatable file name>:]<symbol>
```

The <relocatable file name> may be necessary since identical symbols may occur in separate assemblies. If this is the case, and the <relocatable file name> is not specified, the address of the first symbol encountered in the table will be used. Examples:

```
D#NEGATE
D#TEST1.R:START+4
D#TEST1.R:START+#TEST2.R:START
D#BEGIN*
L 1:TEST1,,#SPACE-10
```

If you use a symbol which is not in the currently loaded symbol table, the Debugger displays the message:

```
UNDEFINED SYMBOL
```

at the bottom of the screen.

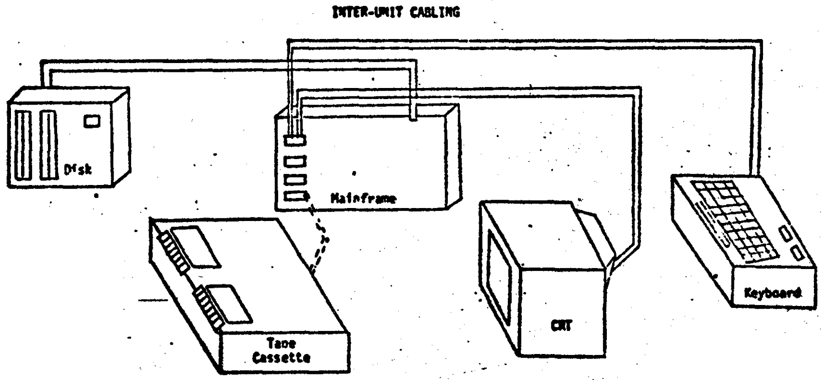
## A P P E N D I X

# FUTUREDATA SYSTEM 31 CHECKOUT

1. Connect the keyboard & CRT to the mainfram.

CAUTION

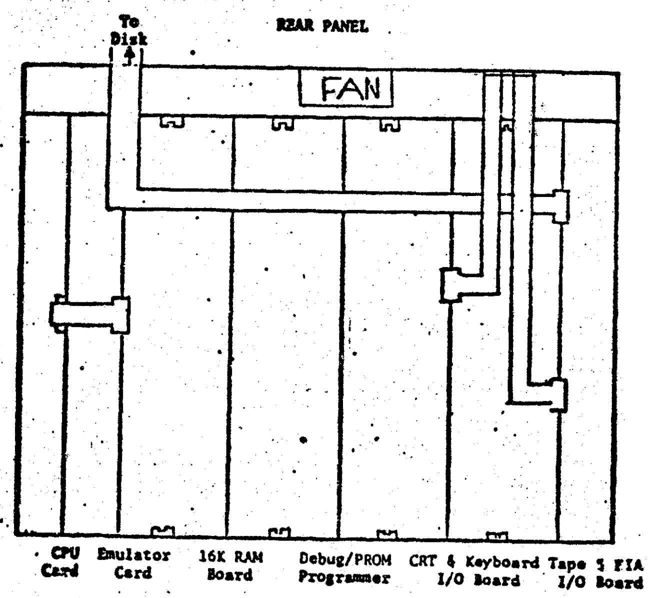
Connecting keyboard into the wrong connector may cause dammage to the keyboard.



2. Open the mainframe top and connect disk drive connector to the rear plug on the TAPE & EIA I/O Board. Ensure that red strip on ribbon cable is toward the rear of the unit. Ensure that all cards are properly seated.

CAUTION

Be careful, if removing CPU card and/or Emulator card. They are interconnected with short ribbon cables which may become disconnected. Both cards should be removed at the same time.



3. Connect CRT and Disk power plugs into the auxillary power sockets on the rear of the mainframe. Then plug in the mainframe to a wall socket.

**CAUTION**

Applying or removing power from the disk drive with a diskette in the unit may destroy the diskette.

4. Turn on the power switches on the mainframe and the disk drive.

5. Insert the system diskette, into DRIVE 0 with the label toward the power switch, and close the diskette door.

6. Press the LOAD button on the keyboard; then type D. The system will initialized with the prompt; FUTUREDATA DISK OPERATING SYSTEM-VER 1.0

7. Type D, return on the keyboard. A directory of the files on the disk will be printed on the CRT.

**CAUTION**

Ensure that the diskette is for the same microporcessor as the CPU. i.e. 8080 diskette, 8080 CPU card.

8. Type JD return. A display of the contents of memory around the location 0000 should be seen.

9. Type JE return. A display of the double dotted line editor, plus a prompt at the bottom of the screen should be seen.

10. Type JA return. A prompt indicating which assembler you have available should be seen.

11. Type JU return. A display of the copy utility should be seen.

12. This completes the verification of all major functions of the Microsystem 31.

13. Open the disk drive door by pressing straight in on the latch to the left of the door. The diskette should be ejected.

14. Remove power from all units, and disconnect them.

## GETTING STARTED WITH THE FUTUREDATA MICROSYSTEM 31

### INITIALIZING A NEW DISK

1. Initialize system by applying power, inserting a system diskette, pressing the LOAD button, and typing D.

2. Place a write enable tape over the write protect slot on the rear edge of a blank double density soft sectored diskette. Then insert it into drive 1 with the label facing to the right.

3. Type I return. The system will respond with the question: INITIALIZE DISKETTE IN DRIVE 1?
   Type Y. Any other response will result in an error indication.

4. When the system is finished initializing the new diskette, the following message will be displayed:

```
FILES ON DRIVE 1:  76 FREE TRACKS
PW 1 DIR
```

### DUPLICATING A DISKETTE

1. First, initialize a diskette as defined above.

2. Then insert the diskette to be copied into drive Ø, leaving the initialized diskette in drive 1.

3. Type X Ø,1,A  return.

DOUBLE DENSITY DISK SYSTEMS

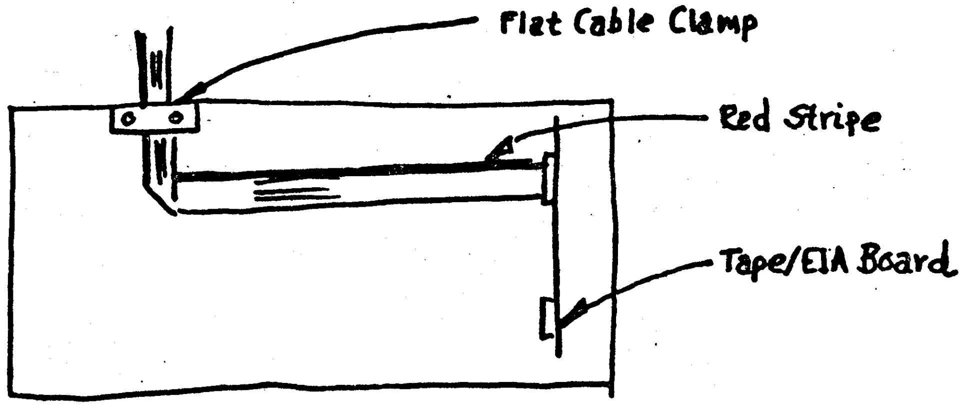
PRELIMINARY

## 1.0 Introduction

The Microsystem/31 with double density dual disk drive provides over 1 megabyte of on-line random access disk storage. The diskettes are organized as 77 tracks of 52 sectors each. Each sector is 128 bytes. The first track on each diskette is reserved for the disk directory. The user may have up to 76 files of one track each per diskette, one file of 76 tracks or any combination in between. The minimum file is 52 sectors of 128 bytes or 6656 bytes total. The maximum file is 76 tracks of 6656 bytes or 505,856 bytes total.

## 2.0 Installation

The Microdisk/3 may be connected to any Microystem. The Microdisk/3 is connected to the Microsystem with a 20 conductor flat ribbon cable. Connect the Microdisk/3 cable to J2 on the Tape/EIA board. This is the only board with two 20 conductor, 3M headers. The red stripe on the cable should be aligned toward the ar of the system and the cable routed out through the flat cable amp at the left rear.



See also section 18.1.2 of "FUTUREDATA DISK OPERATING SYSTEM USERS MANUAL"

It will also be necessary to replace the system bootstrap EPROM. The procedure varies somewhat depending on the CPU card in the Microsystem. If the CPU card is a Z-80, 8085, 6800, 6802 or 8080A mod II; then the bootstrap EPROM on the processor board is eplaced. The double density EPROM's are labled Z80DD for the 80A mod II, 8085, and Z-80. They are labled 68DD for the 6800 d 6802. If the CPU card is an 8080A (p.c. card 10020 rev. A or B) then the following is required.

1. If it is a tape based system (Microsystem/10 or

Microsystem/15) then the bootstrap EPROM on the Tape/EIA board must be disabled as described in 18.1 of "FUTUREDATA DISK OPERATING SYSTEM USERS MANUAL".

2. In addition, tape based systems will require a PROM/RAM board to hold the double density boot EPROM. This is supplied with all Microdisk/3 packages that connect to existing systems and all 8080A systems that do not have the mod II CPU board.

If a Microprinter is also to be used with the system, then the Microprinter is connected to the disk in a daisy chain fashion.

To connect the printer: remove the top cover on the Microdisk/3 and connect the 20 connductor printer cable to the 20 connductor header clamped to the top of drive 1. Note: the red stripes on the the two cables should match. Also note that the Microdisk/3 must be powered on in order to use the printer, if the printer is "daisy chained" through the disk. Under normal operating conditions, this is always the case.

1.0 Memory Strapping

The double density DOS software requires the following memory board strapping:

Systems utilizing 8K RAM boards

| | |
|---|---|
| 16K memory | Connect jumpers at locations 0, and 5 on board 1. Connect jumpers at locations 1, and 6 on board 2. |
| 24K memory | Connect a jumper at location 0 on board 1. Connect jumpers at locations 1, and 5 on board 2. Connect jumpers at locations 2, and 6 on board 3. |
| 32K memory | Connect a jumper at location 0 on board 1. Connect a jumper at location 1 on board 2. Connect jumpers at locations 2, and 5 on board 3. Connect jumpers at locations 3, and 6 on board 4. |
| 40K memory | Connect a jumper at location 0 on board 1. Connect a jumper at location 1 on board 2. Connect a jumper at location 2 on board 3. Connect jumpers at locations 3, and 5 on board 4. Connect jumpers at locations 4, and 6 on board 5. |
| 48K memory | Connect a jumper at location 0 on board 1. |

Connect a jumper at location 1 on board 2. Connect a jumper at location 2 on board 3. Connect a jumper at location 3 on board 4. Connect jumpers at locations 4, and 5 on board 5. Connect a jumper at location 6 on board 6.

Systems utilizing 16K RAM boards

| | |
|---|---|
| 16K memory | Connect jumpers at locations 0, 1, 5, and A6 on board 1. |
| 32K memory | Connect jumpers at locations 0, and 1 on board 1.! Connect jumpers at locations 2, 3, 5, and A6 on board 2. |
| 48K memory | Connect jumpers at locations 0, and 1 on board 1. Connect jumpers at locations 2, and 3 on board 2. Connect jumpers at locations 4, 5, and A6 on board 3. |

Systems utilizing a 32K RAM card.

| | |
|---|---|
| 32K memory | Connect jumpers at locations "5-2", "6-3", and "32". |

Systems utilizing 32K and 16K cards.

| | |
|---|---|
| 48K memory | Connect jumpers at location "32" on the 32K RAM card. Connect jumpers at locations 4,' 5, and A6 on the 16K RAM card. |

Systems utilizing a 48K RAM card.

| | |
|---|---|
| 48K memory | Connect jumpers at locations "5-4", and "48". |

## 4.0 Disk Drive Strapping

Figures 1 and 2 show the option strapping for the double density disk drives. Note the different strapping for drive "0" and drive "1". Some fault isolation can be accomplished by interchanging the functions of drive "0" and drive "1". This is done by removing the rear panel.

CAUTION: POTENTIALLY LETHAL VOLTAGES EXIST INSIDE THE DISK SUBSYSTEM. THE POWER CORD MUST BE DISCONNECTED FROM THE WALL ET WHENEVER THE COVER OR REAR PANEL ARE REMOVED.

he jumpers on the drives may now be altered to change drive "0" into drive "1" and vice-versa. The 34 conductor cable connecting the the two disk drives and disk controller can now be .

disconnected from the suspected drive and the other substituted in its place.