MUSIL

Programming Guide

3600

MUSIL Programming Guide

Revision 1
March 1976
RCSL 42 - i 0344

A/S REGNECENTRALEN
Documentation Department

Author and
Text Editor: Joan Rosenstein

KEY WORDS: Programming, coding, MUSIL, RC 3600, source language.

ABSTRACT: This manual shows how to program an RC 3600 in the MUSIL high-level language.

SUPPORTING DOCUMENT:
RC 3600 MUSIL Programmer's Reference Card (RCSL 42 - i 0355)

Printed by A/S Regnecentralen, Copenhagen

Table of Contents

PART ONE - BASIC SYSTEM ARCHITECTURE

Part One
Basic System Architecture

1.1 Introduction

In this Part One, we shall see how the RC 3600 system is built up and operated. We shall also take a brief overall look at the applications software available on the system.

1.2 The RC 3600 System

The RC 3600 is a system that is composed of hardware and systems software. The hardware is a 16-bit per word mini-computer which is particularly reliable and sturdy, but not optimally flexible nor specifically designed for the support and terminal functions that an RC 3600 system is designed to perform. This flexibility and suitability is provided by the systems software.

The RC 3600 system is a satellite system. That means that it is ultimately associated with a larger host computer. The job of the host computer in an RC 3600/mainframe configuration is to perform computation and data management, often in the form of data base or file management. The job of the RC 3600 in this configuration is to take over those functions that can be separated from the mainframe, such as I/O, peripheral management, data entry and collection, data conversion, and various tasks associated with communications.

The RC 3600 system can be used with or without standard or custom-made applications programs supplied by Regnecentralen. For those users who wish to program their own applications, a special high-level programming language is provided. This language, called MUSIL, is designed for programming support functions. Therefore, it is strong on I/O handling and weak on computational facilities.

The central idea in RC 3600 structure is to complement each element of hardware with systems software. We shall first survey these hardware elements.

1.2.1 <u>RC 3600 Hardware</u>

RC 3600 hardware consists of a central unit with its associated core memory, peripheral units with their associated channels and controllers, a real-time clock, an interrupt system, and a bootstrap loader. With this equipment alone, one can process only one job at a time, each sort of I/O device must have its own data transfer protocol, and high-level programming languages cannot be used. There is also a direct memory access channel for use by the faster peripherals to access core without going through the central processor, but the RC 3600 hardware alone cannot utilize this speed maximally, because only one job can proceed at a time with this hardware, which means that the whole machine is limited in speed by the slower peripherals in I/O-oriented tasks, which are the main tasks of the RC 3600. To solve these problems, various software elements are used to complement each hardware element.

1.2.2 RC 3600 Systems Software

RC 3600 systems software is composed of a monitor, the systems process S, I/O utility routines, and peripheral device drivers.

1.2.2.1 The Monitor has the job of implementing multiprogramming. It does this by complementing the hardware interrupt system with a software interrupt system utilizing a software real-time clock. It provides the means for each process to communicate with the others. It can, therefore, provide a means for several different jobs to be executed at the same time by providing each of them in turn with time slices regulated by the clock. In this way more than one job can run at the same time and slow I/O processes do not slow down the overall performance of the system.

1.2.2.2 The Systems Process S implements a software core allocation system. It creates process descriptors for each process it loads, and these descriptors can be used by the monitor in its task of mediating information among the various processes. S also replaces the buttons and switches that give the operator direct access to the hardware with an operating system that gives the operator access to the system as such.

1.2.2.3 The Drivers replace the individual peculiarities of the various peripheral device types with a single I/O protocol, enabling the monitor to treat I/O processes on the same level with any other processes. Those routines that all the drivers use in common are gathered together in the I/O utility procedures, so as to make each driver as small as possible.

1.3 RC 3600 Applications Software

Most RC 3600 users require one or more standard or custom-made applications programs from Regnecentralen. For such customers full job runs are usually provided on one medium, e.g., on one magnetic tape or in one card deck. These runs contain both the systems and the applications software. Other customers require their own programming capability. For them it is necessary to have a run medium with systems software and a program production package.

1.3.1 Applications Run Media

Those using ready-made programs receive a medium with a monitor, a "basic system", I/O utility routines, drivers, and one or more applications programs in object code, along with an interpreter to execute the object code.

1.3.1.1 Basic Systems consist of a systems process S, a console device driver, and an autoload device driver. The last is necessary because S must have a device driver from which to load application modules. The console device driver can be of two sorts, a driver for the Operator Control Panel (OCP) or a driver for a keyboard device. Basic systems have been developed to keep the space needed for systems software as small as possible.

1.3.2 Program Production Packages

These packages can be supplied separately or on a medium with the necessary systems software, which in this case also must include an interpreter. The package itself consists of a MUSIL compiler, a MUSIL Text Editor, any necessary drivers, and a program generator.

1.3.2.1 The MUSIL Compiler converts MUSIL source code into MUSIL object code, which is executed by a separate piece of software called the MUSIL Interpreter.

The MUSIL compiler can also integrate assembly-coded subroutines, called "code procedures" into the object code output from the compiler. In addition, it can copy parts of MUSIL programs into locations in other MUSIL programs.

1.3.2.2 MUSIL Text Editor allows editing on MUSIL source text by character, string, line, or page. It operates by taking in the source text as data to itself.

1.3.2.3 Program Generators combine one or more compiled MUSIL programs onto a single run medium along with the necessary systems software. They can also integrate into the run "command files", which are files containing code that substitutes for direct operator action, which makes a run more automatic than it would ordinarily be.

1.4 RC 3600 Operation

RC 3600 operation is both simple and flexible. Standard programs can be run automatically by a few simple commands or can be closely controlled by the operator through decision points in the programs that give rise to requests for the input of "run-time parameters" by the operator. Error messages are also complete and easily understood. The programmer who wishes to operate an RC 3600 system should obtain an RC 3600 Operator's Guide and/or an RC 3600 Data Conversion Operator's Reference Card.

Program writing can also be done at the RC 3600 console. Programmers who wish to create or edit MUSIL programs at the machine should also provide themselves with an RC 3600 MUSIL Programmer's Reference Card, which contains information on program writing, compilation, and editing, as well as program production through run generation.

1.4.1 Operation Principles

The operator may communicate with any loaded process, including the systems process S. Messages to and from S are indicated on a keyboard device by CTRL G (BELL) and S and the symbol >, respectively, and on the OCP by the LOAD button and LOAD lamp.

To load a medium, the command LOAD is used, and to select a file from a loaded medium, the command INT, meaning "interpret", is used.

The word "file" is also used to indicate a device whose medium does not contain a catalog. Thus, the paper tape reader is operated as a file. On the other hand, a disc contains, normally, more than one file and a catalog, so that in operating with a disc one must specify which file on the disc one wishes to access. This use of "file" simplifies RC 3600 operation for non-cataloged media.

Most standard programs can be operated from an OCP, but more complicated programs and program writing require a keyboard device. The user of an RC 3600 system with an OCP can request from Regnecentralen one or more command files to make the use of his machine more flexible. For example, with an OCP all programs must be loaded from the autoload device, but with the proper command files another device can be used for program load, imitating this capability of the keyboard devices.

Part Two
The MUSIL Program

2.1 MUSIL Program Structure

MUSIL programs must be written in modular form. Comments may be placed anywhere in the program, as long as they are placed outside of words, numbers, and text strings. (It is, of course, most common to place them between statements or between rows of a table.) Comments are signaled by exclamation points, thus:

! THIS IS A COMMENT !

MUSIL programs are presented in five sections, which must be given in the following order:

The Constant Section is the first section. It is normally present. In it are defined all the constants that will be used in the program, both numerical constants and text string constants. Constants may not be defined anywhere else in a MUSIL program.

The Type Section may be absent. If it is not, then it must be the second section. In it are defined types, or categories, of variables. It is used mostly for file type definitions, for file types are usually long. This section provides only a convenient short-hand type of definition for variable structures. It cannot itself define variables.

The Variable Section is used to define variables of both numerical and text string type, as well as records and files. Variables cannot be defined anywhere else in the program.

The Procedure Section is, properly-speaking, a part of the Main Program Section that follows, but it is convenient to speak about it as a separate section. In it the programmer can define his own procedures, usually I/O exception-handling procedures, that he can call later on from the Main Program Section by name.

The Main Program Section contains the program's instructions.

MUSIL commands can be viewed as of two types: I/O commands and other commands. In this Part Two we shall discuss only the latter. I/O commands are discussed in Part Three.

2.2 The Constant Section

In the Constant Section we define numerical or text string constants as well as tables of numerical or text string constants. Conversion tables are often found in this section, but conversion tables can also be made into separate programs. (Certain conversion table programs can be obtained from Regnecentralen.)

2.2.1 Identifiers

The definition of a constant has the following form:

name = value,

The "name" may consist of any sequence of letters of the alphabet or numerals, but it must begin with a letter and may not include characters other than letters and numbers. The name may be of any length, but only the first 7 characters and the total character count will be used by the program. That is, to the program the following names are the same

MYNAME11 and MYNAME13

but

MYNAME11 and MYNAME111

are different.

A "value" may be a number, a text string, or a table of numbers or text strings.

2.2.2 Numbers

A number may be assigned to an identifier (a "name"). The number may be decimal, octal, or binary; but whatever the number, its binary equivalent must not exceed 16 bits. For decimal numbers this means that the number must fall between

-32768 and +32767

For octal numbers the range is

0 to 8'177777

Octal numbers are generally used to express bit patterns conveniently.

Numbers may have spaces between a sign and the numerals, but there must not be spaces within the number itself. Identification of numbers is done as follows:

```
NUM1 = 2'011001,     ! A BINARY NUMBER !
NUM2 = 8'775,        ! AN OCTAL NUMBER !
NUM3 = -23005,       ! A DECIMAL NUMBER !
```

In the absence of a sign, a positive sign will be assumed.

Decimal points may not be used within numbers.

2.2.3 Text Strings

Strings representing ASCII texts may be assigned a name. This is done according to the following model:

name = 'text string', or
name = "text string", or
name = "text string', or
name = 'text string",

Either single or double quotation marks may be used, and mixing single and double quotation marks within the same identification is also allowed. An example of the identification of a text string might be

ALPHA22 = "ERROR HAS OCCURRED",

After ALPHA22 has been thus defined, it can be used as the name of the text string ERROR HAS OCCURRED.

Text strings cannot be operated on arithmetically. Once they have been named, however, they can be

assigned to a variable as its current value,
used in text comparisons,
output on the operator console,
output to a peripheral device.

Text strings may include byte values given by their numeric representation. One common use of this facility is to output control characters, for example,

ALPHA = '<45>',

places the binary value of decimal 45 into location ALPHA. The angular parentheses indicate a byte value. Since the ASCII code for decimal 45 is a minus sign, ALPHA can now be regarded as containing a textual minus sign.

If we write, for example,

V = "<8'126>",

then the ASCII code for the letter V can be retrieved from location V for later use.

As another example, if we write

CR = "<13>",

then the ASCII code for a Carriage Return can be retrieved from location CR for use in controlling a line printer.

Strings of such ASCII characters can also be placed together under one "name", so that they can be called together as a sequence of ASCII characters. If we write, for example,

ALPHA31 = "<45><0><10>",

then ALPHA31 will contain the ASCII codes for

Minus sign NUL Line feed

No punctuation is used between the factors of such a string of ASCII characters.

2.2.4 Tables of Constants

Tables of numerical or text string constants can be defined in the Constant Section. Tables are in fact text strings themselves, so that their elements cannot be operated on arithmetically in a direct way.

Tables are identified as follows:

name = #element1 blank element2 blank ....#, or
name = "<element1><element2> ....",

where the quotation marks may, again, be single or double, or any mixture of single or double quotation marks. The blanks denote spaces and/or new lines.

As examples we can take the following:

LPTTABLE = #14 0 64 89 56 8'377 0 65#,
LPTTABLE = "<14><0><64><89><56><8'377><0><65>",

which are equivalent definitions for the same table identification.

Note that the following definitions are absolutely equivalent:

ALPHA = #45#, and ALPHA = "<45>",

Table definitions are, therefore, very useful for writing device conversion tables and can include all characters, as well as control characters.

2.2.5 Section Structure

The Constant Section begins with the key word

CONST

which is not followed by any punctuation. Directly after CONST come the desired identifications in any order. Each identification is followed by a comma, except for the last identification, which is followed by a semicolon, thus:

```
CONST
          ALPHA  = 45,
          BETA1  = -8'377,
          GAMMA = "MOUNT TAPE";
```

Because the whole MUSIL program can be regarded as one single compound statement, identifications may be entered with any spacing. Blanks will be ignored, as long as they do not occur within names or values.

2.2.6 Some Cautions On the Use of Constants

Values are stored in their named locations left-justified. Numerical constants are stored as given in the program. String constants are marked by a binary zero at the end of the text. When text string are read out to another location or to a peripheral device, this binary zero is stripped off. If, therefore, a text string is assigned to a variable during program execution and then output on a console device, the console will not stop printing at the end of the text, unless a binary zero has been appended to the text by the programmer.

To avoid such annoying occurrences, a binary zero should be placed at the end of each text string that will be operated on in some way before being output to the console. This is done thus:

```
ALPH = "OUTPUT THIS TEXT<0>",
```

As an example of what else can happen if this is not done, suppose I have assigned the text THIS MESSAGE IS WRONG to ALPHA20. Suppose that later on in the program I wish to change the contents of ALPHA20 to THIS IS ALPHA and then output the contents of ALPHA20. What I want to output is

THIS IS ALPHA

but what I will get is

THIS IS ALPHAIS WRONG

Using the binary zero after the text strings would solve this problem.

String constants that will not be moved around in the program do not, of course, need to conclude with this explicit binary zero.

Text strings may mix ASCII characters and byte values thus:

```
MTTEXT = "<7><10>MT ERROR",
```

The free use of blanks allows long tables to be presented in a way that makes them easy to check. One neat way to insert tables is the following.

```
LPTTABLE = #0   0   0   0   0   0   0   0   0     ! 0th !
            0   0   0   0   0   0   0   0   0     ! 9th !
           33  34  35  36  37  38  39  40  41     ! 18th !
           45  46  47  48  49  50  51  52  53     ! 27th !
           60  61  62  63  64  65#                ! 36th !
```

where the numbers in exclamation points are comments that give the ordinal position of the first number in each line.

2.2.7 Common Errors

The most common errors in the Constant Section are

1)	CONST,	Punctuation after CONST.
2)	ALP = "2"	No comma after an identification.
3)	LASTONE = 5,	Comma after the last definition in the section.
4)	ANY = 7;	Semicolon after an identification that is not the last.
5)	GA.1 = 2,	A name that does not consist of only numbers and letters.
6)	1FIVE = "A",	A name that begins with a number.
7)	TW 00 = 2,	A blank within a name.
8)	J = 3.4,	Presence of a decimal point.
9)	K = 55 6,	Presence of a blank within a value.
10)	L = 2,330,	Presence of a comma within a value.
11)	B = "<8'128>",	Wrong ASCII value.
12)	G = "<7>,<10>",	Comma between ASCII values of a text string.
13)	TABLE = #0,33,37#	Commas within a table.
14)	ALPHA134 = 5, ALPHA137 = 6,	Doubly-defined name (program disregards all characters of a name after the first 7).
15)	HIGH = 50000	Value too large.

2.2.8 Constant Section Example

```
                                                       ! RC36-00007 PAGE 02 !
CONST
NOQ=            8,
OPTXTS=
'<14><6>
<10>PROG NO  :     7<0>
<10>BLOCK NO : <0>
<10>FILE NO  : <0>
<10>REWIND   :   <0>
<10>FIXRECS  :   <0>
<10>MAXCOL   : <0>
<10>MINCOL   : <0>
<10>BLOCKED  : <0>',
START=          'START',
STOP=           'STOP',
SUSPEND=        'SUSPEND',
CONT=           'CONT',
INT=            'INT',
STATE=          'STATE',
MINUS=          '-',
PLUS=           '+',
FIVE=           '<5><0>',
FIFTEEN=        '<15><0>',
NL=             '<10>',
NEXTPARAM=      '<27>',
SP1A=           '<9>',
ENDLINE=        '<13><0>',
RETURN=         '<13>',
RUNTXT=         '<8><4><10>RUNNING<13><0>',
CRTXT=          '<7><10>CR ERROR    ',
MTTXT=          '<7><10>MT ERROR    ',
EOJTXT=         '<14><7><10>END JOB<13><0>',
SUSTXT=         '<7><10>SUSPENDED<13><0>',
CRMOUNTDECK=    '<14><7><10>LOAD CARD DECK<13><0>',
MTMOUNTTAPE=    '<14><7><10>MOUNT DATA TAPE<13><0>',
ENDTAPE=        '<14><7><10>END-OF-TAPE MARK<13><0>',
CONTSTATE=      '<10>CONTSTATE:   <0>',
SPACES=         '
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@',
```

```
                                                                     ! RC36-00007 PAGE 03 !
CRTABLE=              ! CR CONTROLLER FORMAT TO EBCDIC
                  0    1    2    3    4    5    6    7 !
#
!    0 !     64  241  242  243  244  245  246  247
!    8 !    249   49   50   51   52   53   54   55
!   16 !    248  121  122  123  124  125  126  127
!   24 !     56   57   58   59   60   61   62   63
!   32 !    240   97  226  227  228  229  230  231
!   40 !    233   33   34   35   36   37   38   39
!   48 !    232  105  224  107  108  109  110  111
!   56 !     40   41   42   43   44   45   46   47
!   64 !     96  209  210  211  212  213  214  215
!   72 !    217   17   18   19   20   21   22   23
!   80 !    216   89   90   91   92   93   94   95
!   88 !     24   25   26   27   28   29   30   31
!   96 !    208  161  162  163  164  165  166  167
!  104 !    169  225   98   99  100  101  102  103
!  112 !    168  160  170  171  172  173  174  175
!  120 !    104   32  234  235  236  237  238  239
!  128 !     80  193  194  195  196  197  198  199
!  136 !    201    1    2    3    4    5    6    7
!  144 !    200   73   74   75   76   77   78   79
!  152 !      8    9   10   11   12   13   14   15
!  160 !    192  129  130  131  132  133  134  135
!  168 !    137   65   66   67   68   69   70   71
!  176 !    136  128  138  139  140  141  142  143
!  184 !     72    0  202  203  204  205  206  207
!  192 !    106  145  146  147  148  149  150  151
!  200 !    153   81   82   83   84   85   86   87
!  208 !    152  144  154  155  156  157  158  159
!  216 !     88   16  218  219  220  221  222  223
!  224 !    112  177  178  179  180  181  182  183
!  232 !    185  113  114  115  116  117  118  119
!  240 !    184  176  186  187  188  189  190  191
!  248 !    120   48  250  251  252  253  254  255
#;
```

2.3 The Variable Section

While the Constant Section assigns names to values, the Variable Section names locations in core for the values that the program will use. In order to do this, the program must be told what sort of values will fill these locations.

2.3.1 Identifiers

Variables may be integers, text strings, files, or records. The format for a variable definition is

```
name:type;     or     name,name,....,name:type;
```

The restrictions on "name" are the same as for the Constant Section. "Type" is declared as in the following paragraphs.

File and record variable definition will be discussed in Part Three.

2.3.2 Integer Variables

If we wish to declare a location in such a way that integers can be stored in it, then we write

```
name:INTEGER;
```

For example,

```
D:INTEGER;              or
I:INTEGER;              or
NUMBER5:INTEGER;
```

Blanks may be freely used after or before the colon, e.g.,

```
FIRST :   INTEGER;
NEXT :    INTEGER;
```

so as to give the coding a pleasing appearance and make it easy to read.

Allowable numerical values that can go into these locations are between

-32768 and 32767

2.3.3 Text String Variables

Locations can also be declared for text strings. In this case the number of bytes to be reserved must be declared. The format is

name:STRING(number of bytes);

For example,

```
TEXT1         :    STRING(20);     or
TXT,TEXT,A  :    STRING(6);
```

Text strings assigned later on in the program that are smaller than the number of bytes declared for their location will be left-justified. Implicit binary zeroes will not be assigned.

2.3.4 Section Structure

The Variable Section begins with the word

VAR

which is not followed by punctuation. Each declaration in this section ends with a semicolon, including the last declaration.

2.3.5 Some Cautions On the Use of Variables

There are only a few common mistakes in the Variable Section:

An incorrect name
A punctuation error

2.3.6 Variable Section Example

```
                                                        ! RC36-00007 PAGE 04 !
VAR

OPDUMMY:         STRING(2);        ! RUNTIME PARAMETERS !
PROGNO:          INTEGER;
BLOCKNO:         INTEGER;
FILENO:          INTEGER;
REWIND:          INTEGER;
FIXRECS:         INTEGER;
MAXCOL:          INTEGER;
MINCOL:          INTEGER;
BLOCKED:         INTEGER;

OPTEXT:          STRING(20);       ! COMMUNICATION AREA !
OPSTRING:        STRING(20);
OPDEC:           STRING(10);

OPCONT:          STRING(2);        ! INTERNAL VARIABLES !
NEXTCONT:        STRING(1);
GLCONT:          STRING(1);
WBLOCKED:        INTEGER;
ERRORNO:         INTEGER;
MASK:            INTEGER;
TOM:             INTEGER;
SIGN:            INTEGER;
Q:               INTEGER;
PAR:             INTEGER;
LENGTH:          INTEGER;
OPENED:          INTEGER;
P1:              INTEGER;
P2:              INTEGER;
P3:              INTEGER;
S1:              STRING(2);
S2:              STRING(2);
NEXTMT:          INTEGER;
INLENGTH:        INTEGER;
OUTLENGTH:       INTEGER;
CARDSREAD:       INTEGER;
SAVEDSUSPEND:    INTEGER;
```

2.4 The Procedure Section

The Procedure Section has no key word to begin it. It consists only of procedure definitions, one after another.

2.4.1 Defining a Procedure

In the Procedure Section the programmer defines his own procedures. This is done according to the following format:

```
PROCEDURE name;
     BEGIN
          ............. ;
          ............. ;
          .............
     END;
```

Within the BEGIN ..... END described above may be found any of the sort of statements that can be used in the Main Program Section. The Procedure Section is useful for defining what shall occur in the case of an I/O exception situation.

The variables and constants used within the Procedure Section must have been previously defined in their appropriate sections.

2.4.2 Executing a Procedure

To start the execution of a procedure from within the Main Program Section, one writes simply

procedure name;

For example, suppose we have within the Procedure Section

```
PROCEDURE ENDGAME;
     BEGIN
          ............. ;
     END;
```

then, to call this procedure from the Main Program Section, we write

ENDGAME;

simply. Procedures cannot be called by other procedures. before they are defined.

2.4.3 Code Procedures

To incorporate code procedures within a MUSIL program during compilation, the programmer must indicate in his program where the MUSIL compiler is to put these procedures. One does this by writing in the Procedure Section

PROCEDURE name (parameter $specification_1$, ...
..., parameter $specification_5$)
CODEBODY external identification;

The parameter specification and the external identification can be obtained from Regnecentralen.

To call the code procedure from the Main Program Section, one then writes

name ($parameter_1$, ..., $parameter_5$)

Code procedures and instruction in their use are supplied by Regnecentralen.

2.4.4 Some Cautions On the Use of Procedures

The most common errors are

Forgetting to define the procedure's constants and variables in the Constant and Variable Sections, respectively.

Trying to jump from a point within one procedure to a point within another procedure.

Trying to call a procedure from another procedure before the first procedure has been defined.

Trying to call procedures recursively.

2.4.5 Examples of Procedures

Such examples will be given in Part Three.

2.5 The Main Progam Section

The statements that actually control a job are found in the Main Program Section.

2.5.1 Section Structure

The Main Program Section begins with the key word

BEGIN

not followed by any punctuation. It ends with the word

END;

Statements in this section are separated from one another by semicolons. Spaces may be freely used between words.

2.5.2 Arithmetic Operators

MUSIL includes the following arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division

and parentheses may be used freely.

Arithmetic operations are executed from left to right in order of their priority, which is (from high to low)

plus and minus signs
multiplication and division
addition and subtraction

Thus -5+6*7-2/3
is equivalent to -5 + (6 * 7) - (2/3)

2.5.3 Relational Operators

The available relational operators are

<	less than	>=	greater than or equal to
>	greater than	=	equal to
<=	less than or equal to	<>	not equal to

These symbols can also be used for text string comparisons, in which case the strings will be compared lexicographically, that is, on the basis of their numerical ASCII values.

2.5.4 Monadic Operators

There are four monadic operators:

+ number	The plus sign
- number	The minus sign
BYTE textstring	The integer value of the first character of the text string
WORD textstring	The integer value of the first two characters of the text string

For example, suppose TXT contains

"<2'11001001><2'11110011>"

which is equivalent to "A", then

BYTE TXT	gives the integer	0000000011001001
WORD TXT	gives the integer	1100100111110011

2.5.5 Logical Operators

There are three logical operators:

operand 1 AND operand 2

yields the integer value of the logical AND operation as performed on the current value of the two operands

operand 1 SHIFT operand 2

shifts the value of the first operand to the left if the second operand is positive, and to the right if the second operand is negative, shifting the number of bits equal to the numerical value of the second operand; the shift is not cyclical: bits shifted out of the word are lost and the vacant positions are filled with zeroes

operand 1 EXTRACT operand 2 *from right*

extracts bits from the first operand; the number of bits extracted is equal to the current numerical value of the second operand

The operands involved in the logical operators must be integral and the result will be an integer.

For example, let VAR1, VAR2, A, and INT be integer variables, and let the current values be

VAR1	2'0000000010011011
VAR2	2'0000000011100000
INT	2'1111000010111111
A	2'1111000000001111

then

VAR1 AND VAR2	gives	0000000010000000
A SHIFT 2	gives	1100000000111100
A SHIFT (-2)	gives	0011110000000011
INT EXTRACT 8	gives	10111111

2.5.6 Operators In General

The priority from high to low for all operators is

Monadic operators
Multiplying and logical operators
Adding operators
Relational operators

Division of a number by zero, or division of zero by zero, will not give rise to an error message. The result of such operations will be

-1

No indication of integer overflow is given.

When text strings are compared, the comparison will take place only on that number of characters that is the smaller of the two text strings, for example,

> Let ALPHA, declared of length 2, contain TR
> and let BETA, declared of length 5, contain TRANS
> Then, the relation
>
> ALPHA < BETA
>
> could give misleading results, for the comparison will take place only on the first two characters, which are TR in both cases.

Note that the following is not allowed:

"text string" operator "text string"

for operators can operate only on named values.

2.5.7 Assignment

The symbol used for assignment is := so that

A:=B;

assigns the current value of B to A.

Integer values can be assigned directly, thus:

INT1:=5;

as long as INT1 had been declared as an integer variable:

INT1:INTEGER;

But text strings cannot be directly assigned. That is, it is not allowed to write

TEXT3:="REWIND TAPE";

even if TEXT3 had been previously defined in the Variable Section as a text string variable. To put REWIND TAPE into TEXT3, one must have in the Variable Section something like

TEXT3:STRING(11);

and in the Constant Section

T3="REWIND TAPE",

and then one can write in the Main Program Section

TEXT3:=T3;

Then REWIND TAPE will be in TEXT3.

Variables defined as integers may have only integers assigned to them. And variables defined as text strings may have only text strings assigned to them. If, for example, INT1 is an integer variable and TEXT2 is a string variable, then one may not write either

INT1:=TEXT2; or TEXT2:=INT1;

Multiple assignments are not allowed. That is, one may not write

INT1,INT2:=0; or INT1:=INT2:=0;

When text strings are assigned, the number of characters that are moved is equal to the lesser of the number of characters defined for the variables involved. That is, if in the Variable Section we have

TEXT1:STRING(10);
TEXT2:STRING(20);

and in the Main Program Section we have

TEXT1:=TEXT2; or TEXT2:=TEXT1;

then, only ten characters will be moved in either case. In the first case only the first 10 characters of TEXT2 will be moved into TEXT1. In the second case the 10 characters in TEXT1 will be moved into the ten left-most positions of TEXT2, leaving the remainder of TEXT2 unchanged.

2.5.8 Labels

MUSIL statements may be labeled, so as to identify them. The label must be a unique numeric value between 0 and 65535. The format for labeling is

```
label: statement;
```

For example, `35: GAMMA:=5;`

labels the statement GAMMA:=5; with the identifying label 35.

The use of spaces before or after the label's colon is optional.

Note that the statement

```
label: END;
```

must be preceded by a semicolon.

2.5.9 Compound Statements

The sub-statements of compound statements are set off by a BEGIN ... END combination. Thus:

```
BEGIN
        statement 1;
        statement 2;
        ..........
        statement n
END;
```

There is no punctuation after the BEGIN or before the END (unless the END is labeled). Spaces may be freely used to improve readability.

BEGIN ... END phrases may be nested up to the number of 30.

2.5.10 Unconditional Branching

This is represented by the statement

```
GOTO label;
```

with no space within the GOTO.

If the GOTO statement is used to go to a labeled END statement, then the statement before the END statement must conclude with a semicolon, thus:

```
        .............
        .............
        GOTO 60;
        .............
        .............
        ALPHA:=40;
60:     END;
```

No error is committed, however, if the semicolon is used also in other cases.

2.5.11 Conditional Branching

MUSIL has the usual

IF relation THEN statement,

as well as an

IF relation THEN statement_1 ELSE statement_2

If the relation is true, then statement_1 will be executed. If not, then control will pass to the next statement. The IF-relation may be any allowable relation, and the THEN-statement may be any allowable statement, including compound statements. For example,

```
IF ALPHA=5  THEN
   BEGIN
              IF BETA<10  THEN  U:=0;
              IF GA>= 9   THEN  V:=1;
              ...................
   END;
```

The relational expression may contain only variables or constants. The following are not allowed.

```
IF "TEXT1"="TEXT2"  THEN ...      or
IF TEXT="REWIND"  THEN ...
```

With respect to compound statements, note that in our example above, that if ALPHA was not 5, then program flow would have passed to the first statement after the END statement, bypassing the intermediate IF ... THEN statements.

2.5.12 Repetitive Statements

There are two repetitive statements in MUSIL:

```
WHILE ... DO
REPEAT ... UNTIL
```

The format of the first is

```
WHILE relation DO statement;
```

This instruction allows the repetition of an operation as long as the relational statement remains true, e.g.,

```
WHILE X>Y DO
   BEGIN
          ............. ;
          ............. ;
          .............
   END;
```

There is no punctuation after the DO. In the above example if X is never greater than Y, then the DO statement will never be executed.

The format of the second is

```
REPEAT statement UNTIL relation;
```

An example for this command might be

```
REPEAT
          BEGIN
          ............. ;
          ............. ;
          .............
          END
                 UNTIL X=Y;
```

There is no punctuation after REPEAT or before UNTIL.

In this example, if X is equal to Y when END is reached, then the statement will be executed once.

2.5.13 Commands For the Operator Console

MUSIL contains 10 commands that are useful for communications between the program and the operator console. They are called Standard Procedures. Six of them will be described here, and the rest in Part Three.

OPMESS(string variable name);

outputs the text string contained in the string variable or constant named to the operator console. It continues to output the contents of the variable or constant until a binary zero <0> is reached. At most 80 bytes will be output in this way. If the text is less than 80 bytes and does not end in a binary zero, then the output will continue for the full 80 bytes anyway, outputting whatever is in core following the text desired. Text to be output by OPMESS should be in ASCII code.

Example: OPMESS(ALPHA); where ALPHA contains a text string, will output the value of ALPHA on the operator console. Thus,

```
CONST
   ALPH="REWIND";
BEGIN
   OPMESS(ALPHA)
END
```

will output REWIND on the operator console.

OPIN(string variable name);

is the reverse operation. It allows the operator to insert a text string of up to 80 bytes into a variable previously defined as STRING in the Variable Section. In order to give the operator time to input this text, OPIN must be followed by

OPWAIT(integer variable name);

which makes the system wait for the operator's input. The number of characters that the operator actually inputs will be stored in the integer variable by the system.

Example:

```
VAR
   LENGTH:INTEGER;
   MAGTEXT:STRING(80);
BEGIN
   OPIN(MAGTEXT);
   OPWAIT(LENGTH)
END;
```

will allow the operator to put up to 80 bytes of text into the variable represented by MAGTEXT, while the OPWAIT(LENGTH) will give the operator time to do this and will store the number of characters input in LENGTH.

Operator input will normally be terminated, when a control key is used (CR, LF, ESCAPE, etc.).

If OPIN has been used, but it is desired to see if a text has actually been accepted for input from the console, then the command

OPTEST

can be written. If a text has been accepted for input, then OPTEST will give a non-zero value. If the OPIN operation has been unsuccessful, then the value of OPTEST will be zero.

Example:

```
VAR
   LENGTH:INTEGER;
   MAGTEXT:STRING(80);
BEGIN
   OPIN(MAGTEXT);
   WHILE OPTEST = 0 DO
         BEGIN
         ............. ;
         .............
         END;
   OPWAIT(LENGTH);
END;
```

will allow the operator to insert a text into MAGTEXT. If the input is successful, then OPTEST will be non-zero at some point and the WHILE ... DO statement will cease execution.

OPTEST is a standard function. Its current value can be used profitably to control program branching with respect to whether or not operator action has occurred, for example

```
IF OPTEST=0 THEN PSTOP;
```

where PSTOP is a user-defined procedure.

The RC 3600 system operates in binary. Thus all decimal numbers that are input by the operator with OPIN must be converted to binary before they can be used by the machine. This is done with

DECBIN(decimal value name, binary value name);

There must have been defined in the Variable Section two variables. One will be used to store the number inserted in decimal by the operator. The other will be used to store its binary equivalent. The variable with the binary equivalent is the one that must be used for all subsequent MUSIL statements that work with this inserted value.

Example:

```
VAR
   DEC:STRING(10);             ! NB: DECIMAL INPUT IS!
   LENGTH,BIN:INTEGER;         ! DEFINED AS A TEXT STRING!
BEGIN
   OPIN(DEC);
   OPWAIT(LENGTH);
   DECBIN(DEC,BIN);
   IF BIN=0 THEN PSTOP;
END;
```

The decimal value being converted must have no sign. It will be converted into a 16-bit binary value. There will be no check for overflow, so that the number must be less than or equal to 32767. If a non-numeric character appears within the input, then conversion will proceed up to that point and then stop. A plus or a minus sign or a decimal point is considered to be a non-numeric character.

BINDEC(binary value name, decimal value name);

is used with OPMESS. It takes the binary value and stores its ASCII equivalent concluding with a binary zero byte. The decimal value can then be output to the operator console by OPMESS.

Example:

```
VAR
   DEC:STRING(6);
   BIN:INTEGER;
BEGIN
   BIN:=2'1001;
   BINDEC(BIN,DEC);
   OPMESS(DEC)
END;
```

will output decimal 00009 to the operator console for inspection.

The decimal value variable must be defined in the Variable Section as a STRING with a minimum of 6 bytes. If it must be output with a sign, then the sign must be defined separately:

```
CONST
   PLUS="+";
VAR
   DECSIGN:STRING(1);
   DEC:      STRING(6);
   BIN:      INTEGER;
BEGIN
   BIN:=2'1001;
   BINDEC(BIN,DEC);
   DECSIGN:=PLUS;
   OPMESS(DECSIGN);
   OPMESS(DEC)
END;
```

which will output first + and then 00009 on the operator console.

(The binary value will be converted to exactly five decimal digits.)

To output +9 directly, one can make use of the instructions

MOVE and INSERT

The MOVE and INSERT commands are, thus, very useful in connection with commands to the operator console, but they are most often used for I/O. Therefore, they will be discussed in Part Three.

2.5.14 Some Cautions For the Main Program Section

The most common errors relating to the above material are

1) Forgetting a BEGIN or putting punctuation after it.
2) Forgetting the semicolon after an END.
3) Forgetting a semicolon after a statement that precedes a labeled END.
4) Forgetting the parentheses around a negative (i.e., right-hand) SHIFT.
5) Forgetting that bits are lost when SHIFT is used.
6) Mis-sequencing operators.
7) Dividing by zero inadvertently.
8) Comparing text strings of unequal length and forgetting that the comparison will be only on the lesser number of characters.
9) Trying to operate on text strings instead of on their names.
10) Forgetting the colon in the assign symbol.
11) Forgetting to define variables and/or constants before using them.
12) Trying to assign text strings to integer variables or integers to text string variables.
13) In assigning texts to string variables longer than the text, forgetting to take care of the excess text remaining from a previous text assignment.
14) Forgetting the colon after a label.
15) Spelling GOTO as two words.
16) Using constants within relational statements instead of variable or constant names.
17) Illogical entries into compound statements.
18) Forgetting that the REPEAT ... UNTIL statement will always be executed at least once.
19) Forgetting to convert binary numbers to decimal when the decimal value is to be output by OPMESS.
20) Forgetting the OPWAIT instruction.
21) Forgetting to convert decimal numbers input by the operator to binary before trying to use them.
22) Trying to use plus or minus signs with DECBIN.
23) Trying to input a number greater than 32767 from the operator console.
24) Forgetting the semicolon after a procedure name.
25) Using illogical jumps.
26) Trying to use values assigned within a procedure before the procedure has been activated.
27) Forgetting to define a procedure's constants and variables in the Constant and Variable sections.

Part Three
I/O Commands

3.1 Overview

In this section we shall examine the most important MUSIL commands, the I/O commands. I/O commands deal with the physical transfer of data from core to a peripheral device, or from a peripheral device to core. Two kinds of operations are involved in this sort of data transfer.

Control operations do not result in any direct transfer of data, but they are necessary for data transfer. Typical control operations are, for example, the opening or closing of a file, the positioning of a magnetic tape, etc.

Transput operations call for the actual data output or input. Such operations are, thus, of two types: input or output mode. It is the output data that is the purpose of the entire data processing operation.

Both control and transput operations are performed in conformity with messages sent to the appropriate driver process. As noted in Part One, the driver always reports on the success or failure of an I/O operation, that is, it reports on its status.

The status of an I/O operation is put into a status word which is accessable by the programmer as well as by the system itself. The status word tells which aspects of the I/O operation have been successful and which have failed. If a failure is such that data processing can not proceed without some special exception handling procedure, then one of two things can happen.

If the programmer has not provided his MUSIL program with an applicable exception handling procedure, then the system will stop the processing of the job and display an error message on the operatior device to inform the operator of what has occurred. Such error messages have the form

device name ERROR error number

For example

LPT ERROR 21

which means that processing can not continue because the line printer is offline.

After the operator has corrected the situation, the job must be restarted from the beginning.

If the programmer has provided his program with an applicable exception handling procedure, then control will pass to it. Such procedures are called GIVEUP procedures and they are called when the status word is compared with a programmer-generated GIVEUP mask that tells the system which exception situations will be handled by the program instead of by the system.

The status of an I/O operation is only one of the components of the messages that pass between an I/O device driver and the program and system via the monitor. Other information that is passed along concerns the identification of the I/O device and data and constant reporting on where the data is at any time. This information is accessable to the programmer at any time via the file descriptors, which provide names for the locations from which the information can be accessed by the programmer.

I/O operations in MUSIL are arranged in such a way that the programmer is able to avoid all housekeeping tasks associated with I/O, while still retaining the option to assume control of some of these tasks if he so desires. This in fact is the reason why MUSIL was created, it having been felt that no existing programming language fully satisfied this objective in a convenient and logical way. For example, MUSIL provides for an automatic transfer of data between the peripheral device and the buffers, and it also provides the programmer with instructions by which he may assume control of this function. The instructions that give the programmer such direct control are called primitive operations.

Primitive operations are those operations that high-level instructions use to perform their functions. In MUSIL the primitive operations dealing with buffer control are available also to the programmer, though in normal situations he would have no need for them.

3.2 The Organization of Data

I/O data is organized into groups called bytes, records, blocks, and files. This organization represents a hierarchy of data organization that makes it possible to deal with as much data at a time as any given I/O situation allows.

3.2.1 Bytes

For the RC 3600 bytes are groups of 8 bits. They correspond to "characters" which are defined as letters of the alphabet, numerals, or special symbols, for example, punctuation. Most work with MUSIL is done using bytes composed of bit patterns from the ASCII code, but for data any code may be used in working with an RC 3600.

The RC 3600 is a 16-bit per word machine; thus, each word in the machine has room for two bytes. MUSIL contains commands that allow the transput of data byte by byte.

3.2.2 Records

Records are groups of bytes. The concept of a record is a logical idea. There is no way to define a "record" in a way that covers all possible record types, but a record is usually considered to be the smallest piece of information that is of interest to the end-user.

Records are components of larger sets of information, called "files". Records within a file can be of several types.

Fixed length records are the components of a file that consists of records which all have the same number of bytes in them. Variable length records are components of a file in which the length of the records varies from one record to another.

3.2.3 Blocks

Some data media allow records to be blocked, that is, grouped together. Blocking is a physical concept. A block of data is a quantity that is read into memory or written out of memory in one physical operation. Blocking involves some physical delineation on some medium, such as an interblock gap of a magnetic tape or a control character input to a line printer within the data that it receives from memory. On the slower peripheral devices one can observe the occurrences of blocks, for when the end of a block is reached one can often see the device pause for a moment. On some media blocks can also be directly observed. A punched card, for example, is usually treated as a one-record block.

Both fixed and variable length records may be grouped together in blocks. The size of blocks is usually determined by the programmer, whose decision depends on the parameters of the application he is programming for.

Record length format may also be classified as undefined. This means that each record will be treated as a separate block. This is done, for example, when the block size of a magnetic tape input file is unknown to the programmer who must process the tape.

The programmer's decision on blocking strategy is related to his decision on the number of "buffers" that he will employ. Buffers are sections of core that are reserved to hold input or output data for transput from or to peripheral devices. They will be more fully discussed below, but here it should be noted that programmer decisions with respect to blocking, buffer size and number of buffers are very crucial for the speed of an I/O operation. For example, the common situation in which a magnetic tape containing print line images is to be printed out on the RC 3632 line printer operating at 1800 lpm requires 7 buffers for the print lines and one buffer for the magnetic tape input for optimal throughput. Such information can be calculated by the programmer, or it can be provided to him by Regnecentralen on the basis of experience.

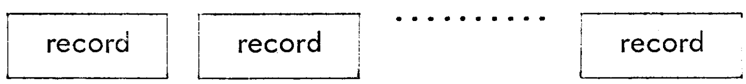
3.2.3.1 Blocking methods. There may be any number of records in a block, including fractions of records, but the most normal cases are

1) To have one record per block, and
2) To have an integral number of records per block.

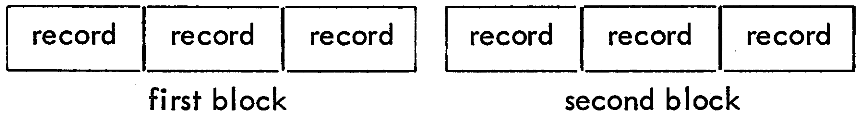
In the first case we say that the records are "unblocked". In the second case we say that the records are "blocked". In many cases the last block of a blocked file of records may not be completely filled with information.

Six types of records are common:

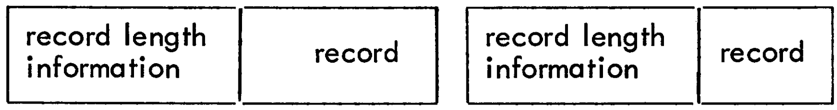
1) Fixed length unblocked, where all the records have the same length and there is one record per block.



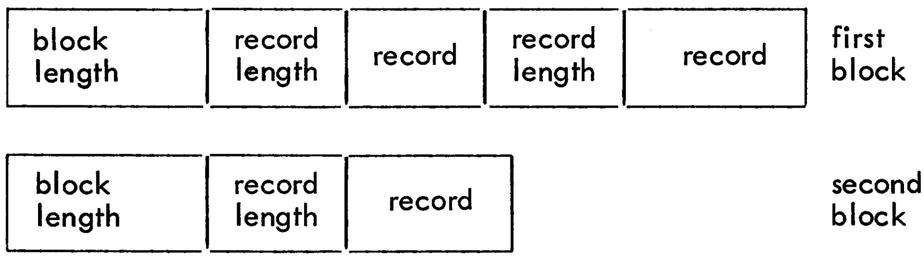
2) Fixed length blocked, where all the records have the same length and are grouped in blocks.



3) Variable length unblocked, where record length varies and there is one record per block.

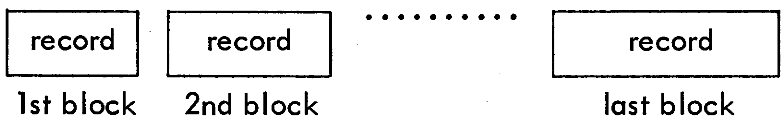


4) Variable length blocked, where record length varies and records are grouped in blocks.

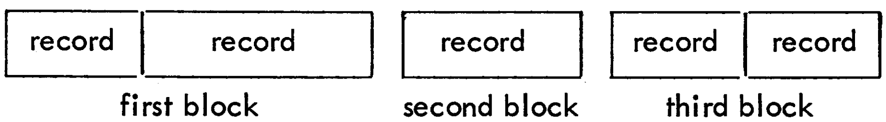


There may be any number of blocks and of records in a block.

5) Undefined length unblocked, where there is one record per block but there is no information about their length.



6) Undefined length blocked, where there is no information about the record length but records are grouped in blocks.



3.2.4 Files

In MUSIL we define a "file" as a set of data stored on some device. For this reason we can use the device name as the file name, if the medium has no catalog. If there is a catalog on the medium, as is usually the case for a disc, for example, then we cannot use the device name directly as the file name.

This usage of the concept of a file allows for simpler programming for non-cataloged media, the common situation on systems without disc support.

3.3 File Descriptors

We have stated that each file in a MUSIL program is defined by a file descriptor. The file descriptor gives the structure of the file and the nature of the data in it. The file descriptor also contains current information on the condition of the data in the file that are undergoing processing. The file descriptor allows both the system and the programmer to know and control what is going on during I/O operations.

Three kinds of information are found in a file descriptor:

1) file identification
2) control information
3) buffer information

File identification includes

1) The name of the file.

    As explained above, uncataloged media are considered to contain but one file. Thus, in this case the name of the file is the name of the device on which the file appears. For example, the file on the paper tape reader is called PTR.

2) The kind gives information about the sort of device the file is on. The kinds include

    Is the medium blocked? Is the device positionable? Is automatic error recovery wanted? For example, for magnetic tape it is usual to repeat a read or write operation that was not successful the first time, but one might specify a non-repeatable kind for a magnetic tape file in certain situations to save read time, or to validity-check tapes.

3) The mode gives information about the data in the file. The mode contains an "operation code" that says if the data is input or output data.

4) Whether conversion is to be done, and if so, a reference to the conversion table.

For character-oriented devices (such as paper tape and card equipment and line printers), conversion is most commonly done during I/O, so that the conversion will most frequently be handled by the driver process. For block-oriented devices, converion must be done in the MUSIL program.

Control information

1) The position of the file. For magnetic tapes this would be the "file number" in IBM usage, that is, the number of tape marks that have been passed. The position also includes the block number when relevant.

2) The status tells whether or not an I/O operation has been successful, and if not, why not.

Buffer information includes

1) Buffer size. How big they are in bytes.

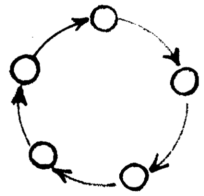
2) Used buffer. Which buffer is currently in use and the size of the block in it (in bytes).

From the above it can be seen that the file descriptor contains two types of information: permanent information identifying the file and its nature, and information on the current state of the file.

The information in the file descriptor can be accessed by the programmer through a system-defined record called FILEDESC. The use of this record will be described below.

3.4 Buffer Strategy

The MUSIL programmer is responsible for defining the number and size of the buffers to be used. The buffers so defined will then constitute a cyclical buffer pool.



At any time during I/O processing one of the buffers will be actually transputting data. This is the used buffer. If something goes wrong during data transfer, in the absence of a programmer-defined GIVEUP procedure within the MUSIL program the system will stop the job and display a device error message to the operator, as explained above. We shall call this the standard procedure in the description below.

Buffer strategy proceeds as follows:

3.4.1 Input

When a file is OPENed for data input to memory the system responds to the OPEN command by making one of the buffers the used buffer, by establishing a pointer from the file descriptor to this buffer. Block length in the file descriptor is set to zero also. The following operations then occur:

1) Input of the first block of the file is started into the used buffer.
2) Input of the second block is started into the second buffer and it is made the used buffer.
3) The process is continued until there are no more free buffers.
4) The first buffer is checked to see if data transfer to it was successfully completed.
5) If not, the standard procedure or a GIVEUP procedure is followed, if so, then the first buffer becomes the used buffer again, and data can be processed in it, after which it is used for more input.
6) This process is followed then for the second buffer, for the third, and so on, until all the data in the file is input.

The buffer pool can be viewed as a continually cycling wheel of buffers. This allows processing to proceed while data is being input. The optimal input speed is achieved when the number and size of the buffers are such as to allow continuous input at the maximum speed that the device allows.

It should be noted that if an input file is CLOSEd during the input process, then the buffer wheel will keep turning to empty the buffers, but the data remaining at the time of CLOSing will be lost.

3.4.2 Output

When a file has been OPENed to receive output, the buffer pool is similarly activated.

1) The first block of output is put into the first buffer, which is the used buffer.
2) The rest of the buffers are filled with output, each becoming the used buffer in its turn.
3) The data transfer from the first buffer to the device is checked for success.
4) If unsuccessful, then the standard procedure or a GIVEUP procedure is followed, and if successful, then the first buffer is ready to receive more data from core and becomes the used buffer.
5) The process is repeated for all the buffers and the "wheel" turns until the external file is complete.
6) The file can now be CLOSEd. E.g., on magnetic tape a double tape mark is written.

3.5 Exception Handling

There are two kinds of "exceptions": errors, such as parity errors, and normal stopping points for operations, such as reaching the end of a tape. Exceptions are reported to the system via the status word in the file descriptor.

After the attempted execution of a driver process operation the success or failure of the operation is reported in the status word. If all the bits of the status word are zero, then the operation has encountered no exceptions and processing can proceed. Though the specific events represented by the bits of the status word are different for different devices, the general over-all representation is as follows:

Bit	Interpretation	Action
0	disconnected	hard error
1	off-line	hard error
2	device busy	if device kind is repeatable, the operation will be repeated; if not, then there is a hard error
3	device mode 1	ignored
4	device mode 2	ignored
5	device mode 3	ignored
6	illegal	hard error
7	End of File	hard error
8	block error	as for bit 2
9	data late	as for bit 2
10	parity error	as for bit 2
11	end of medium	hard error
12	position error	hard error
13	rejected	hard error
14	timer error	hard error
15	(not processed)	ignored

The standard system response to an exception is to try to repeat the operation. It does this up to five times, after which the error is a hard error. Such errors cause the program to stop running and an error message to be displayed on the operator console. The error message consists of the device name and an error number. This number is equal to 20 plus the number of the bit involved. For example, a line printer that is disconnected at the time the system wants to print with it will give rise to the message

LPT ERROR 20

for the name of a line printer is LPT and the status bit involved is number 0, and 20 plus 0 equals 20.

Consult the RC 3600 Data Conversion Operator's Reference Card for the various device error numbers and their interpretations.

The programmer may, however, elect to write a procedure that bypasses some or all of these standard exception-handling facilities of the driver involved. He does this by "turning off" those bits whose actions he wishes to bypass by specifying a

3.5.1 GIVEUP Mask

The GIVEUP mask is formed by creating a constant word that has 1's in those bit positions that correspond to the bits of the status word that the programmer wishes to prevent from initiating the standard system response. For example, if the programmer wishes to write his own exception handling routine for the occurrence of a position error, then his GIVEUP mask must contain a 1 in its bit 12.

GIVEUP masks are usually written in binary for easy readability, for example,

2'1100001111111111

GIVEUP masks are part of the corresponding file descriptor and are associated with

3.5.2 GIVEUP Procedures

Such procedures provide the routine that the system is to follow for each exception condition that the programmer has signified his desire to control by the 1's in the GIVEUP mask.

The programmer need not, however, provide for the standard repetition of the operation, as the GIVEUP procedure will be consulted by the system only after it has tried up to five repetitions of the operation that gave rise to the error.

3.6 Record and File Variables

Record and file variables are defined within the Variable Section. We shall now describe how to do this.

3.6.1 Record Variables

Locations may be reserved for assignment for records. The general format is

```
name:  RECORD
       record structure
       END;
```

For example, if we write

```
VAR
      TWOPARTS:  RECORD
                 HEAD:STRING(4);
                 TAIL:STRING(4)
                 END;
```

then we have reserved space in location TWOPARTS for the assignment of records of 8 bytes, with the first four bytes being given the name HEAD and the last four bytes being given the name TAIL. Subsequent to this definition HEAD and TAIL can be considered to be defined with respect to TWOPARTS, but not defined in themselves. Thus, later on in the Main Program section, HEAD and TAIL are defined only when named together with TWOPARTS, thus

```
TWOPARTS.HEAD          and
TWOPARTS.TAIL
```

The program will not become confused if the same sub-names are used for parts of other records. That is, given

```
TWOPARTS.HEAD          and
INPUTLINE.HEAD
```

the program will know that parts of different records are being referred to.

Further refinements in record definition are also possible. Consider the following:

```
VAR
      S:    STRING(1);
      INREC:     RECORD
                 CCW:S;
                 TEST:S;
                 LINE:STRING(132);
                 STOPF:STRING(2) FROM 1
                 END;
```

This coding defines S as a one-character string location. It defines INREC as a record with the first two bytes as text strings of one byte each. The next 132 characters are called, collectively, LINE and are also text.

Furthermore, STOPF is defined as the name for the first two characters of the record INREC, these two characters being taken together. Thus, later on in the program we can make assignments to either of the first two characters of INREC individually, using their names CCW and TEXT, or we can assign values to them together, using their common name STOPF, viz., via

```
INREC.STOPF
```

Observe that in general each definition within a record definition ends with a semicolon, except that no semicolon is used after RECORD or before END.

As a final example, suppose that instead of giving the first two characters of INREC a name STOPF, we wanted to give a name to the first ten characters of LINE. Then, instead of writing

```
STOPF:STRING(2) FROM 1
```

we would write

```
NEWNAME:STRING(10) FROM 3
```

Note, then, that the characters of a record definition are numbered up from 1. Note particularly that a record may not be longer than 256 bytes.

3.6.2 File Variables

File variables are also defined in the Variable section. The most general format for a file variable definition is

```
name:   FILE
        'device',
        kind,
        number of buffers,
        buffer size,
        type;
        CONV conversion table name
        OF RECORD
                record structure
        END;
```

The meaning of the parts of the file variable definition are

1) Device is the name for the device the file is, or will be, on. Allowable code names are

MT0, MT1, MT2, MT3	Magnetic tape units on the first magnetic tape channel
MT4, MT5, MT6, MT7	Magnetic tape units on the second magnetic tape channel
LP0, LP1	Line printers
SP1, SP2	Serial printers
CDR, RDP	Card reader, Card reader punch
PTR, PTP	Paper tape reader, Paper tape punch
FD0, FD1	Flexible disc drive
DKP0, DKP1	Disc cartridge drive
PLT	Incremental plotter
CT0, CT1	Cassette tape unit
CP0	Charaband printer

The device name must be enclosed in quotation marks.

2) Kind gives information about the device. Allowable kinds are derived from the binary representation of the kind. The binary representations are

bit 15	character-oriented
14	block-oriented
13	positionable
12	repeatable (automatic error recovery)
11	cataloged medium (i.e., disc)

Thus, for a block-oriented, positionable, and repeatable device (such as magnetic tape), the kind word would be

2'0000000000001110

which in decimal is 14, so that kind here would be 14.

3) Number of buffers is selected by the programmer.

4) Buffer size is the size of a single buffer. This is also determined by the programmer.

5) Type refers to the record format. Allowable types are

UB	undefined, blocked	
U	undefined	
F	fixed	
FB	fixed, blocked	
V	variable	(IBM V format)
VB	variable, blocked	(IBM VB format)

6) GIVEUP procedure name need not occur. It is present if the programmer wishes to define his own device error routines. The procedure name must, of course, refer to a user-defined procedure in the Procedure section.

7) GIVEUP masks occur together with GIVEUP procedures. Inserting a 1 bit in any position in the word will cancel the corresponding action in the system's standard error routine.

8) Record structure can be given in terms of the name of a previously-defined record type (cf. 4.1), or can be specified directly, in the same way that records were defined, thus:

```
TYPE
        PLINE    = RECORD P:STRING(50)  END;
VAR
        IN       : FILE
                   'MT0',14,1,60,FB
                   OF PLINE;
```

or

```
VAR
        IN       : FILE
                   'MT0',14,1,600,FB
                   OF STRING(50);
```

3.6.3 Example of File Definitions

```



IN:     FILE                          ! INPUT FILE DESCRIPTION        !
        'MTO',                        ! NAME OF INPUT DRIVER          !
        14,                           ! KIND= REPEATABLE,             !
                                      !       POSITIONABLE,           !
                                      !       BLOCKED.                !
        2,                            ! BUFFERS                       !
        1995,                         ! BUFFERSIZE                    !
        FB;                           ! FIXED BLOCKED                 !

        GIVEUP
        MTINERROR,                    ! MT ERROR PROCEDURE            !
        2'0110001111111110            ! GIVE UP MASK                  !
        OF RECORD                     ! RECORD STRUCTURE              !
           CCW:         STRING(1);
           DATA:        STRING(132)
        END;



OUT:    FILE                          ! OUTPUT FILE DESCRIPTION       !
        'LPT',                        ! NAME OF OUTPUT DRIVER         !
        2,                            ! KIND= BLOCKED                 !
        5,                            ! BUFFERS                       !
        133,                          ! BUFFERSIZE                    !
        U;                            ! UNDEFINED                     !

        GIVEUP
        LPERROR,                      ! LP ERROR PROCEDURE            !
        2'1100001011110110;           ! GIVE UP MASK                  !

        CONV
        LPTTAB                        ! CONVERSION TABLE              !

        OF RECORD                     ! RECORD STRUCTURE              !
           CCW:         STRING(1);
           DATA:        STRING(132)
        END;
```

3.7 Using the File Descriptor

The file descriptor contains a great deal of information about its corresponding file. The programmer may from time to time wish to access this information.

3.7.1 Accessing the File Descriptor

The most common use of the file descriptor is in accessing the status word for use in a GIVEUP procedure, but it is also common to use the file descriptor for displaying information about the file. To enable the programmer to have easy access to the file descriptor, there is a system-defined record for each file that is defined as follows:

```
FILEDESC = RECORD
   ZNAME    : STRING(6);    file name
   ZMODE    : INTEGER;      operation mode
   ZKIND    : INTEGER;      file kind
   ZMASK    : INTEGER;      GIVEUP mask
   ZFILE    : INTEGER;      file position
   ZBLOCK   : INTEGER;      block position
   ZCONV    : INTEGER;      conversion table address
   ZFORM    : INTEGER;      record format
   ZREM     : INTEGER;      number of bytes remaining in the
                            current block
   ZLENGTH  : INTEGER;      record length
   ZFIRST   : INTEGER;      address of the first byte of the
                            current record
   ZTOP     : INTEGER;      address of the top byte of the
                            current record (that is, the first
                            byte of the next record)
   Z0       : INTEGER;      the status word
   ZUSED    : INTEGER;      address of the used buffer
   ZSHAREL  : INTEGER       block length for the buffer
   END;
```

To access the current value of any field of the file descriptor, one writes

```
filename.field
```

For example,

```
IN.Z0
```

signifies the status word of the file called IN. To access this status word, then, we must have a command something like

```
STATIN:=IN.Z0;
```

which will put the status word into the previously-defined integer variables STATIN, or we could write directly something like

```
IF IN.Z0 = 2'1100000000000000 THEN
...........................
```

To display the current block number, for example, of the current file, one could write

```
BINDEC(IN.ZBLOCK,OUTPUT);
OPMESS(OUTPUT);
```

where OUTPUT had been previously defined as a string variable.

3.7.2 Examining the Status Word

The status word can be examined by writing

```
BINDEC(filename.Z0,string variable name);
OPMESS(string variable name);
```

but since this event occurs frequently, a special command is available for displaying the contents of the status word on the operator device.

```
OPSTATUS(filename.Z0,string name);
```

In order to use this command, we must in the Constant Section define a string constant that will be capable of outputting a text for each bit of the status word that is non-zero. This is best illustrated by an example.

Let us define a constant called ERRORS, thus:

```
CONST
        ERRORS  =  "DISCONNECTED  <10><0>
                    OFF-LINE      <10><0>
                    BUSY          <10><0>
                    BYTE OR NOISE <10><0>
                    HARDWARE      <10><0>
                    WRITE RING    <10><0>
                    UNIT RESERVED <10><0>
                    EOF           <10><0>
                    BLOCK SIZE    <10><0>
                    OVERRUN       <10><0>
                    PARITY        <10><0>
                    EOT           <10><0>
                    POSITION      <10><0>
                    DRIVER        <10><0>
                    DENSITY       <10>   ";
```

which is the interpretation of the error messages for a magnetic tape unit arranged so that each diagnostic will be printed on a separate line, that is <10> is a line feed and <0> is a carriage return.

Then, the OPSTATUS command

```
OPSTATUS(filename.Z0,ERRORS);
```

will display on the operator device the lines of ERRORS that correspond to positions of the status word that are non-zero. That is, if the status word contains at the time of inquiry

```
1000000000000000
```

then

```
DISCONNECTED
```

will be displayed, and a line feed and carriage return will be accomplished.

3.7.3 Example of a GIVEUP Procedure

```
                                                    ! RC36-00099 PAGE 11 !


        PROCEDURE MTINERROR;
        BEGIN
            IF IN.Z0 AND 256 <> 0 THEN GOTO 9;  ! EOF !
            IF IN.Z0 AND 8'041000 = 0 THEN BLOCKNO:= IN.ZBLOCK;
            IF IN.Z0 SHIFT 1 < 0 THEN OPMESS(MTMOUNTTAPE);
            IF IN.Z0 SHIFT 1 >= 0 THEN
            BEGIN
                OPMESS(MTTXT);
                MASK:=IN.Z0;
                SHOWERROR;
            END;
            REPEAT OPSTOP UNTIL STOPPED <> 0;
            IF STOPPED = 1 THEN GOTO 1;
            OPMESS(RUNTXT);
        END;

        PROCEDURE LPERROR;
        BEGIN
            NEXTLP:= OUT.Z0 AND 8'000020;
            OUT.Z0:=OUT.Z0 - NEXTLP;
            IF OUT.Z0 SHIFT 1 < 0 THEN
            OUT.Z0:=OUT.Z0 AND 8'041342;
            IF OUT.Z0 = 8'040000 THEN IF NEXTLP <> 0 THEN
            OUT.Z0:= NEXTLP;
            IF OUT.Z0 AND 8'001342 <> 0 THEN
            OUT.Z0:= OUT.Z0 AND 8'001342;
            IF OUT.Z0<>0 THEN
            BEGIN
                OPMESS(LPTXT);
                MASK:=OUT.Z0;
                SHOWERROR;
                NEXTLP:=0;
                REPEAT OPSTOP UNTIL STOPPED <> 0;
                IF STOPPED = 1 THEN GOTO 1;
                OPMESS(RUNTXT);
                IF OUT.Z0 AND 8'141362 <> 0 THEN
                REPEATSHARE(OUT);
            END;
        END;
```

Recommended GIVEUP procedures for many situations are available from Regnecentralen.

3.8 Accessing File Contents

Once a file has been defined in the Variable Section and opened in the Main Program Section, its records can be read into memory or output to a medium.

When a record has been read into memory, it may be accessed for assignment or comparison with respect to the data it contains. Such operations, however, can take place only on previously-defined (that is, named) records or parts of records. The format for accessing the data in a file is

filename↑

for accessing the data as a whole, and

filename↑ .fieldname

for parts of a record.

For example,

```
VAR
          ALPHA:STRING(2);
          IN:  FILE
               "MT0",14,1,1340,FB
               OF RECORD
                         CCW       :STRING(1);
                         SELECT1   :STRING(1)  FROM 1;
                         DATA      :STRING(2)  FROM 2
               END;
BEGIN
          ...............
          ...............
          ALPHA:=IN↑ .DATA;
          ...............
          ...............
END;
```

puts the current contents of DATA from file IN into ALPHA. Open and get record statements are defined below.

3.9 I/O Commands

Though buffer strategy in MUSIL is based on the transfer of blocks of data, actual processing is almost always performed on records or characters. There are two levels of I/O commands in MUSIL, therefore: the higher-level record and character commands and the primitive procedures that the higher-level commands utilize.

We shall now explain the I/O commands in detail.

3.9.1 Opening and Closing Files

OPEN(filename,mode);

This command ensures that file identification is established, reserves the peripheral device involved, prepares for conversion if necessary, and initializes the file. The "mode" is input or output and the conditions of transput, for example, odd or even parity. It is represented by a decimal number that varies from device to device. Operation mode numbers can be found in the Appendix.

CLOSE(filename,release);

closes the file. For input files the closing process concludes all pending data transfers, but it does not check the data transfer, since this data will be lost anyway. For output files the closing process completes and checks all pending data transfers and writes a terminator to the file, for example, a file mark in the case of magnetic tape. "Release" may be any integer. If it is zero, then the device will not be made available to any other program and the device will not be automatically set off-line. If "release" is not zero, then the device will be set off-line. For example, in the case of magnetic tape, release not equal to zero will cause the tape to be rewound and the tape unit to be set off-line.

3.9.2 Record-by-Record Data Transfer

This is the most common transput means in MUSIL. It can be done after a file has been OPENed.

GETREC(filename, variable name);

This command has the general effect of making "the next" record available for processing. The first time it is used after an OPEN command on the file, it must start the transfer of data into the buffer wheel and establish control over the turning of the wheel. It does this by calling the primitive commands INBLOCK, TRANSFER, and WAITTRANSFER (explained below). When used subsequently, it simply makes the next record in the block the "current" record, so that this record can be processed. In this way GETREC can be used to step through the records in the buffers until it again becomes necessary to input a block of data. The specific actions of GETREC in various situations are best described by an example.

Let us examine the following situation:

```
VAR
        SIZE:     INTEGER;
        INFILE:   FILE
                  ...............      !FILE DEFINITION!
                  END;
BEGIN
        OPEN(INFILE, mode);
        ...............
        GETREC(INFILE,SIZE);
        ...............
        ...............
END;
```

The GETREC command would then give rise to the following:

<u>For file record format undefined and unblocked, U</u>:

The general effect is that the contents of a buffer become available for processing. If all the buffers are empty, then the instruction starts the transfer of data into the buffer wheel as well. Let us say we are using cards and have only one buffer, then the effect of the instruction is to read a card. For paper tape and one buffer, the effect is to read and process as much of the tape as will fit into one buffer. The size of the block read is put into SIZE.

For file record format undefined and blocked, UB:

The general effect is to make a number of characters available for processing, this number being equal to the current value of SIZE. If the buffers are empty at the time of call, then SIZE characters must be input. If there is less than SIZE characters in the buffer, then the next block is input until SIZE characters can be available for processing. This situation can be taken advantage of for reading the first character of a tape block to see what kind of tape block one is dealing with. For example, if SIZE and MT0 have been previously defined, and we write

```
OPEN(MT0, mode);
SIZE:=1;
GETREC(MT0,SIZE);
IF BYTE MT0 = some binary code  THEN
        BEGIN
        SIZE:=MT0.ZREM;
        GETREC(MT0,SIZE);
        ...............
        ...............           !PROCESSING OF BLOCK!
        END;
```

then the effect is to read and examine the first byte of the tape and then decide if we want to read in the rest of the first block.

For file record format fixed length and unblocked, F:

In this case the record format has been defined in the Variable Section, as has its length. The command will make a record available, reading in a record, if necessary. The number of bytes made available will be put into SIZE. This will be the record length in this case. In case a record of the correct length cannot be gotten, the exception procedure is called with status bit 8 (block length). If the error is accepted, the record is delivered as a short block.

For file record format fixed length and blocked, FB:

The instruction will make a record available. If it cannot find one in the buffers, then it will first begin to read the next blocks into the buffer wheel. The record length will appear in SIZE. Incorrect record lengths are handled as for F formats.

For file record format variable length and unblocked, V:

The next record will be made available. If the buffers are empty, input will proceed into the buffer wheel. The first four bytes of the record, which contain the record length, are decoded and put into SIZE. The variable length format used is the IBM V format. Incorrect record lengths are handled as for F formats.

For file record format variable length and blocked, VB:

This is the means of handling the IBM VB format. The next record is made available, by reading the next blocks into the buffer wheel, if necessary. The first four bytes of a new block, containing the block length, are decoded. In any case the first four bytes of the record, containing the record length, are decoded and placed in SIZE.

PUTREC(filename,name or number or expression);

is the command that makes space for a record available for output and starts the buffer wheel turning for output by calling the primitive procedures OUTBLOCK, TRANSFER, and WAITTRANSFER. The specific actions are

For file record format undefined and unblocked, U:

PUTREC(FILENAME,SIZE);

A previous buffer is output, then space is reserved in the buffer for the next record to be output the next time PUTREC is called. The name, number, or expression is the length of the record.

For file record format undefined and blocked, UB:

The command makes space for the next record of SIZE bytes in the buffer. If there is no room for it, it outputs a buffer and reserves space in the new buffer. The name, number or expression in this case is the length of the record.

For file record format fixed and unblocked, F:

The number, name, or expression is ignored. The record length is that given in the record definition. The effect of the command is to output a previous record and make space for the current record in a new buffer.

For file record format fixed and blocked, FB:

If the record can fit into a buffer, it is put there; if not, a block will be output to make room for it. The name, number or expression in the command is ignored. The SIZE is that of the record definition.

For file record format variable and unblocked, V:

This command utilizes the IBM V format. A block will be output to make room for a new record. The block size and record size (which are equal) are computed, so that the output medium will be in IBM V format. The record of length equal to the name, number or expression will have room in the space reserved.

For file record format variable and blocked, VB:

If there is no room in the buffer, a block is output to make room for the current record. In any case the record descriptor is computed and the block descriptor is up-dated. The record size must be in the name, number, or expression.

Some general notes on GETREC and PUTREC should be made. First, the second factor of the command has different general functions in the two expressions. For GETREC, it relates to the length of the record coming in from the buffer, a given fact. In PUTREC on the other hand, the record length must be given to the program so that it knows how much output space to look for in the output buffer wheel (except for F formats).

Secondly, GETREC and PUTREC are not equivalent to READ and WRITE, respectively. They do not, in fact, move data.

3.9.3 Character-by-Character Data Transfer

Data transfer by character can be done on files that have records which are undefined and unblocked. The general effect of character I/O commands is to input or output data character by character.

INCHAR(filename, integer variable name);

takes the next available byte from the input buffers and places it in "integer variable name". If the buffers are all empty, then INCHAR will call INBLOCK, TRANSFER, and WAITTRANSFER, in order to read data from the input device.

OUTCHAR(filename, value);

checks to see if there is room for one byte in the output buffers. If not, then it calls OUTBLOCK, TRANSFER, and WAITTRANSFER, in order to make room for the byte. The byte that is put into the buffer is the last byte of the "value", which may be a number, the current contents of a variable, or the value of an expression.

OUTTEXT(filename, string name);

outputs the string contained in the "string name". Output continues until a binary zero is reached. Thus, the string to be output must contain such a binary zero. This command is useful, for example, for putting text onto printout. OUTTEXT is a shorthand command offered in MUSIL as a convenience. Its effect can be obtained by using combinations of MOVE and OUTCHAR.

3.9.4 Primitive Procedures

The normal MUSIL I/O commands are arranged in a hierarchy in which the commands higher up in the hierarchy operate by calling on the commands lower down:

highest level:	GETREC	PUTREC	INCHAR	OUTCHAR
intermediate level:	INBLOCK		OUTBLOCK	
lowest level:	TRANSFER	WAITTRANSFER		

The intermediate and lowest level commands are "primitive procedures" when they are used directly and explicitly by the programmer. It should be noted that few situations will arise in which it will be necessary for the programmer to use them explicitly, while the higher-level commands always use them.

TRANSFER(filename, length, operation mode);

is the command that actually starts the transfer of physical data to and from the peripheral devices. In the input mode TRANSFER causes the used buffer to be filled from the data medium and moves the used buffer pointer to the next buffer in the buffer wheel. In output mode TRANSFER writes the contents of the used buffer to the medium and moves the used buffer pointer to the next buffer in the output buffer wheel. The number of bytes transput is in "length", which must be an integer expression. It should be noted that TRANSFER does not look to see if the used buffer is ready to receive or output new data. Therefore, TRANSFER is always followed by

WAITTRANSFER(filename);

which looks to see if the used buffer is ready with data input or finished with output. If not, it delays the transput until the used buffer is ready.

INBLOCK(filename);

administrates the data input operation. Its effect is to call TRANSFER until all the buffers are started in the buffer wheel. Then it calls WAITTRANSFER to prepare the first input.

OUTBLOCK(filename);

administrates the data output operation. Its effect is to call TRANSFER to start the output buffers. Then it calls WAITTRANSFER in order to ready the next buffer.

REPEATSHARE(filename);

can be used only within a GIVEUP procedure. This command restarts a rejected operation and then returns to the internal waittransfer, so that the operation can be completed before returning control to the next command in the program.

3.10 Data Manipulation

In this section we shall explore some commands that are used for the direct manipulation of data. These commands are understandable both as simple MUSIL commands and as I/O handling commands. They facilitate the solution of data conversion problems.

```
WAITZONE(filename);
```

is mentioned here because it is a lower-order command that is used by the following command, as well as being useful in general to the programmer. Its effect is to halt processing in an orderly way, so that processing can later be resumed without trouble. For input files it empties the buffer wheel, skipping data. For output it assures that all output operations have been completed. For example,

```
IF operator action is called for THEN
      BEGIN
         WAITZONE(filename);
         ...............        !OPERATOR ACTION!
         IF ready now THEN
         ...............        !RESUME PROCESSING!
      END;
```

```
SETPOSITION(filename, file number, block number);
```

allows one to position a positionable medium, such as a magnetic tape. SETPOSITION calls WAITZONE first, in order to halt processing. Then it positions the medium to the desired block. For example,

```
SETPOSITION(MT0,3,8);
```

positions the tape to the eighth block of the third file on the tape on tape unit 0. (Here we use the word "file" in its physical sense.)

The command

MOVE (string name, from n+1st byte, to string name, from n+1st byte, for number of bytes)

that can be used for operator communication can also be used for data manipulation, for example,

MOVE(IN↑,1,OUT↑,0,LENGTH);

will take the current record of file IN, starting with the second byte, and move it into the current record from file OUT, starting with the first byte, until LENGTH number of bytes have been moved. Note that if LENGTH is too big, there will be no error message.

MOVE <u>cannot</u> be used to move bytes around within a single string.

CONVERT(string name, string name, table name, length);

can be used to convert between media. It is best explained by an example.

Consider the expression

CONVERT(MT0↑,OUT↑,TABLE1,OUT.ZLENGTH);

This command would take the current record of file MT0 and convert it according to the table defined as TABLE1, and put the result into the current record of file OUT, doing this for as many bytes of the first record as is represented by OUT.ZLENGTH, which is precisely the length of the current record of file OUT.

"Length" can be an expression, a number, or a variable name.

INSERT(byte name, record name, place);

This instruction can insert a byte into a place in a string. For example,

INSERT(VALUE,OUT↑,2);

places the 8 least significant bits of VALUE into the third byte position of the current record of file OUT.

3.11 Possible I/O Errors

It is impossible to list all the things that might go wrong where I/O commands occur. In the list that follows, we have tried to list the most obvious places where the programmer might take extra care.

1) In setting up a GIVEUP mask remember to start counting bits from zero and be aware of what each bit signifies for the specific device you are using.
2) Try to provide for restart capacity in all operational situations where operator error might occur, such as his forgetting to put a line printer on line.
3) Check carefully for correct punctuation in record and file definitions.
4) Make sure you have up-to-date information on device names and kinds.
5) For difficult decisions about buffer size and number, you may consult Regnecentralen for advice.
6) For difficult GIVEUP procedures, particularly in communications programs, you may consult Regnecentralen.
7) Be particularly careful in using the GETREC and PUTREC commands. They are a common source of error.
8) Be particularly careful when using the primitive procedures. They are very sensitive.
9) Remember that MOVE cannot be used to move bytes within a string.

3.12 Example of a MUSIL Program

```
!                                                    RCSL:   43-GL140

                                                     AUTHOR: CT

                                                     EDITED: 74.08.12

                          PROGRAM RC36-00007.00

                          MUS CARDS TO TAPE

KEYWORDS:            MUSIL,CONVERSION,CDR,MTA,LISTING

ABSTRACT:            THIS PROGRAM HANDLES 80-COLUMN CARDS IN EBCDIC CODE
                     AND GENERATES FIXED OR VARIABLE LENGTH FORMAT RE-
                     CORDS WHICH MAY BE WRITTEN WITH A SPECIFIED NUMBER
                     OF RECORDS IN EACH OUTPUT BLOCK.
                     OUTPUT IS EBCDIC CODE IN BLOCKS OF UP TO 2000 BYTES
                     ON NO LABEL TAPE WITH OR WITHOUT BLOCK AND RECORD
                     LENGTH FIELDS DUE TO THE RECORD TYPE SPECIFICATION.
                     THE PROGRAM MAY BE OPERATED FROM EITHER OCP OR TTY.

RCSL 43-GL141:       ASCII SOURCE TAPE         !
```

```
!                                                       RC36-00007 PAGE 01

TITLE:               MUS CARDS TO TAPE.

ABSTRACT:            THIS PROGRAM HANDLES 80-COLUMN CARDS IN EBCDIC CODE
                     AND GENERATES FIXED OR VARIABLE LENGTH FORMAT RE-
                     CORDS WHICH MAY BE WRITTEN WITH A SPECIFIED NUMBER
                     OF RECORDS IN EACH OUTPUT BLOCK.
                     OUTPUT IS EBCDIC CODE IN BLOCKS OF UP TO 2000 BYTES
                     ON NO LABEL TAPE WITH OR WITHOUT BLOCK AND RECORD
                     LENGTH FIELDS DUE TO THE RECORD TYPE SPECIFICATION.
                     THE PROGRAM MAY BE OPERATED FROM EITHER OCP OR TTY.

SIZE:                5564 BYTES. INCLUDING TWO 80 BYTES INPUT BUFFER
                     AND ONE 2000 BYTES OUTPUT BUFFER.

DATE:                AUGUST 12TH 1974.

RUNTIME PARAMETERS:
    BLOCK NO : 00001      NEXT BLOCK TO BE WRITTEN TO CURRENT FILE.
    FILE NO  : 00001      THE FILE IN WHICH THE BLOCK IS WRITTEN.
    REWIND   :     +      INDICATES IF REWIND OF TAPE AT END OF INPUT.
    FIXRECS  :     +      INDICATES OUTPUT RECORD FORMAT FIXED/VARIABLE.
                          NOTE: THIS PARAMETER CAN ONLY BE CHANGED BEFORE
                          THE PROGRAM IS STARTED AND AFTER END OF JOB.
    MAXCOL   : 00080      MAXIMUM NUMBER OF COLUMNS TRANSFERRED WHEN
                          VARIABLE LENGTH FORMAT OUTPUT OR NUMBER OF
                          COLUMNS WHEN FIXED LENGTH FORMAT OUTPUT.
    MINCOL   : 00080      MINIMUM NUMBER OF COLUMNS TRANSFERRED WHEN
                          VARIABLE LENGTH FORMAT OUTPUT.
    BLOCKED  : 00025      MAXIMUM NUMBER OF RECORDS IN EACH BLOCK.
OTHER OUTPUT MESSAGES:
    CONTSTATE:  +/-       STATE OF CONTINUE SWITCH (TTY ONLY).
    PROG NO  :      7     PROGRAM EXECUTION IS STOPPED.
    RUNNING               PROGRAM EXECUTION IS STARTED.
    SUSPENDED             DRIVERS RELEASED, PROGRAM EXECUTION IS STOPPED.
    LOAD CARD DECK        CARD READER HOPPER EMPTY AND CONTINUE IS ON.
    CR ERROR NNNNN        CONSULT THE RC3600 OPERATORS MANUAL.
    MT ERROR NNNNN        CONSULT THE RC3600 OPERATORS MANUAL.
    END JOB               PROGRAM EXECUTION IS TERMINATED.

INPUT MESSAGES:
    STOP                  STOPS EXECUTION WRITING PROG NO  :     7.
    SUSPEND               STOPS EXECUTION RELEASING DRIVERS (TTY ONLY).
    INT                   NEXT PARAMETER IS DISPLAYED
                          (ESCAPE BUTTON ON TTY HAS SAME EFFECT).
    STATE                 ALL PARAMETERS ARE DISPLAYED (TTY ONLY).
    "VALUE"               CURRENTLY DISPLAYED PARAMETER IS CHANGED
                          TO "VALUE".
    "TEXT"="VALUE"        THE PARAMETER IDENTIFIED BY "TEXT" IS
                          CHANGED TO "VALUE"
    CONT                  STATE OF CONTINUE SWITCH IS INVERTED.
    START                 PROGRAM EXECUTION IS STARTED.
                          NOTE: AFTER CR ERROR START MEANS ACCEPTING
                          THE ERRONEOUS INPUT, AFTER MT ERROR START
                          MEANS REPEATING THE WRITE OPERATION.
SPECIAL REQUIREMENTS:     NONE                                      !
```

```
CONST

NOQ=            8,

OPTXTS=
'<14><6>
<10>PROG NO  :      7<0>
<10>BLOCK NO : <0>
<10>FILE NO  : <0>
<10>REWIND   :   <0>
<10>FIXRECS  :   <0>
<10>MAXCOL   : <0>
<10>MINCOL   : <0>
<10>BLOCKED  : <0>',

START=          'START',
STOP=           'STOP',
SUSPEND=        'SUSPEND',
CONT=           'CONT',
INT=            'INT',
STATE=          'STATE',
MINUS=          '-',
PLUS=           '+',
FIVE=           '<5><0>',
FIFTEEN=        '<15><0>',
NL=             '<10>',
NEXTPARAM=      '<27>',
SP1A=           '<9>',
ENDLINE=        '<13><0>',
RETURN=         '<13>',

RUNTXT=         '<8><4><10>RUNNING<13><0>',
CRTXT=          '<7><10>CR ERROR    ',
MTTXT=          '<7><10>MT ERROR    ',
EOJTXT=         '<14><7><10>END JOB<13><0>',
SUSTXT=         '<7><10>SUSPENDED<13><0>',
CRMOUNTDECK=    '<14><7><10>LOAD CARD DECK<13><0>',
MTMOUNTTAPE=    '<14><7><10>MOUNT DATA TAPE<13><0>',
ENDTAPE=        '<14><7><10>END-OF-TAPE MARK<13><0>',
CONTSTATE=      '<10>CONTSTATE:   <0>',
SPACES=         '
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@',
```

```
                                                                        ! RC36-00007 PAGE 03 !
CRTABLE=            ! CR CONTROLLER FORMAT TO EBCDIC
                 0    1    2    3    4    5    6    7 !
#
!    0 !        64  241  242  243  244  245  246  247
!    8 !       249   49   50   51   52   53   54   55
!   16 !       248  121  122  123  124  125  126  127
!   24 !        56   57   58   59   60   61   62   63
!   32 !       240   97  226  227  228  229  230  231
!   40 !       233   33   34   35   36   37   38   39
!   48 !       232  105  224  107  108  109  110  111
!   56 !        40   41   42   43   44   45   46   47
!   64 !        96  209  210  211  212  213  214  215
!   72 !       217   17   18   19   20   21   22   23
!   80 !       216   89   90   91   92   93   94   95
!   88 !        24   25   26   27   28   29   30   31
!   96 !       208  161  162  163  164  165  166  167
!  104 !       169  225   98   99  100  101  102  103
!  112 !       168  160  170  171  172  173  174  175
!  120 !       104   32  234  235  236  237  238  239
!  128 !        80  193  194  195  196  197  198  199
!  136 !       201    1    2    3    4    5    6    7
!  144 !       200   73   74   75   76   77   78   79
!  152 !         8    9   10   11   12   13   14   15
!  160 !       192  129  130  131  132  133  134  135
!  168 !       137   65   66   67   68   69   70   71
!  176 !       136  128  138  139  140  141  142  143
!  184 !        72    0  202  203  204  205  206  207
!  192 !       106  145  146  147  148  149  150  151
!  200 !       153   81   82   83   84   85   86   87
!  208 !       152  144  154  155  156  157  158  159
!  216 !        88   16  218  219  220  221  222  223
!  224 !       112  177  178  179  180  181  182  183
!  232 !       185  113  114  115  116  117  118  119
!  240 !       184  176  186  187  188  189  190  191
!  248 !       120   48  250  251  252  253  254  255
#;
```

```

VAR

OPDUMMY:          STRING(2);       ! RUNTIME PARAMETERS !
PROGNO:           INTEGER;
BLOCKNO:          INTEGER;
FILENO:           INTEGER;
REWIND:           INTEGER;
FIXRECS:          INTEGER;
MAXCOL:           INTEGER;
MINCOL:           INTEGER;
BLOCKED:          INTEGER;

OPTEXT:           STRING(20);      ! COMMUNICATION AREA !
OPSTRING:         STRING(20);
OPDEC:            STRING(10);

OPCONT:           STRING(2);       ! INTERNAL VARIABLES !
NEXTCONT:         STRING(1);
GLCONT:           STRING(1);
WBLOCKED:         INTEGER;
ERRORNO:          INTEGER;
MASK:             INTEGER;
TOM:              INTEGER;
SIGN:             INTEGER;
Q:                INTEGER;
PAR:              INTEGER;
LENGTH:           INTEGER;
OPENED:           INTEGER;
P1:               INTEGER;
P2:               INTEGER;
P3:               INTEGER;
S1:               STRING(2);
S2:               STRING(2);
NEXTMT:           INTEGER;
INLENGTH:         INTEGER;
OUTLENGTH:        INTEGER;
CARDSREAD:        INTEGER;
SAVEDSUSPEND:     INTEGER;

```

```
                                                  ! RC36-00007 PAGE 05 !

IN:     FILE                          ! INPUT FILE DESCRIPTION !
        'CDR',                        ! NAME OF INPUT DRIVER !
        2,                            ! KIND= BLOCKED !
        2,                            ! BUFFERS !
        80,                           ! SHARESIZE !
        U;                            ! UNDEFINED !

        CONV
        CRTABLE;                      ! CONVERSION TABLE !

        GIVEUP
        CRERROR,                      ! CR ERROR PROCEDURE !
        2'0110001011110110            ! GIVE UP MASK !

        OF STRING(80);                ! RECORD STRUCTURE !



OUT:    FILE                          ! OUTPUT FILE DESCRIPTION !
        'MTO',                        ! NAME OF OUTPUT DRIVER !
        14,                           ! KIND= REPEATABLE, !
                                      !       POSITIONABLE, !
                                      !       BLOCKED. !
        1,                            ! BUFFERS !
        2000,                         ! SHARESIZE !
        FB;                           ! FIXED(BLOCKED)/VARIABLE(BLOCKED) !

        GIVEUP
        MTOUTERROR,                   ! MT ERROR PROCEDURE !
        2'0110011111011011            ! GIVE UP MASK !

        OF STRING(80);                ! RECORD STRUCTURE !

```

```
PROCEDURE INITPOSITION;
BEGIN
    IF FIXRECS= -1 THEN
    BEGIN
        IF IN.ZMODE=33 THEN FIXRECS:=-2;
        IF IN.ZMODE=0  THEN OPEN(IN,21);
    END;
    IF FIXRECS= -2 THEN
    BEGIN
        IF IN.ZMODE=21 THEN FIXRECS:=-1;
        IF IN.ZMODE=0  THEN OPEN(IN,33);
    END;
    IF FIXRECS=-1 THEN OUT.ZFORM:=3;
    IF FIXRECS=-2 THEN OUT.ZFORM:=5;
    IF OUT.ZMODE=0 THEN OPEN(OUT,3);
    IF BLOCKNO=OUT.ZBLOCK THEN
    IF FILENO=OUT.ZFILE THEN
    GOTO 999;
    WAITZONE(IN);
    SETPOSITION(OUT,FILENO,BLOCKNO);
999:
END;

PROCEDURE CONTINUE;
BEGIN
    GLCONT:=OPCONT;
    OPCONT:=NEXTCONT;
    NEXTCONT:=GLCONT;
    OPMESS(OPCONT);
END;
```

```
PROCEDURE DIRECTUPDATE;
BEGIN
    P1:=0;          ! INDEX IN INPUT STRING !
    P2:=0;          ! INDEX IN CONSTANT STRING !
    P3:=1;          ! PARAMETER NUMBER IN CONSTANT STRING !
    REPEAT BEGIN
        MOVE(OPTEXT,P1,S1,0,1);
        MOVE(OPTXTS,P2,S2,0,1);
        WHILE BYTE S1 <> BYTE S2 DO
        BEGIN
            IF BYTE S2 = 0 THEN P3:=P3+1;
            P2:=P2+1;
            MOVE(OPTXTS,P2,S2,0,1);
            IF P3>NOQ THEN S2:=S1;
        END;
        IF P3<=NOQ THEN
        BEGIN
            WHILE BYTE S1 = BYTE S2 DO
            BEGIN
                P1:=P1+1;
                P2:=P2+1;
                MOVE(OPTEXT,P1,S1,0,1);
                MOVE(OPTXTS,P2,S2,0,1);
                IF BYTE S1 = 61 THEN
                BEGIN
                    MOVE(OPTEXT,P1+1,OPTEXT,0,10);
                    LENGTH:=LENGTH-P1-1;
                    Q:=P3;
                    MOVE(OPDUMMY,Q*2,OPDUMMY,0,2);
                    PAR:= WORD OPDUMMY;
                    P3:=NOQ;
                END;
            END;
            P2:=P2-P1+1;
            P1:=0;
        END;
    END UNTIL P3>=NOQ;
END;
```

```
          PROCEDURE OPCOM;
          BEGIN
1000:         Q:=0;
1010:         REPEAT BEGIN
                  IF OPTEXT=STATE THEN
                  BEGIN Q:=1; OPMESS(CONTSTATE); IF OPCONT=FIVE THEN
                      OPMESS(PLUS); IF OPCONT=FIFTEEN THEN
                      OPMESS(MINUS); GOTO 1040;
                  END;
1015:             Q:=Q+1;
1020:             OPSTATUS(1 SHIFT(16-Q),OPTXTS); IF Q<>1 THEN BEGIN
                  MOVE(OPDUMMY,Q*2,OPDUMMY,0,2);
                  PAR:= WORD OPDUMMY;
                  IF PAR = -1 THEN OPMESS(PLUS);
                  IF PAR = -2 THEN OPMESS(MINUS);
                  IF PAR >= 0 THEN
                  BEGIN BINDEC(PAR,OPDEC); OPMESS(OPDEC); END; END;
                  IF OPTEXT=STATE THEN GOTO 1060;
1040:             OPMESS(ENDLINE);
                  OPWAIT(LENGTH);
                  OPTEXT:=OPSTRING;
                  OPIN(OPSTRING);
                  IF OPTEXT=STATE THEN BEGIN Q:=0; GOTO 1015; END;
                  IF OPTEXT = SUSPEND THEN
                  BEGIN
                      SAVEDSUSPEND:= 1;
                      GOTO 1040;
                  END;
                  IF LENGTH > 6 THEN DIRECTUPDATE;
                  IF LENGTH > 6 THEN GOTO 1020;
                  IF OPTEXT = START THEN GOTO 1070;
                  IF OPTEXT = STOP  THEN GOTO 1000;
                  IF OPTEXT = CONT THEN
                  BEGIN CONTINUE; GOTO 1040; END;
                  IF OPTEXT = INT THEN GOTO 1060;
                  IF OPTEXT = NEXTPARAM THEN GOTO 1060;
                  IF OPTEXT = NL THEN GOTO 1020;
                  IF OPTEXT = ENDLINE THEN GOTO 1020;
                  IF OPTEXT = RETURN THEN GOTO 1020;
                  SIGN:=0;
                  IF OPTEXT = MINUS THEN SIGN:=-1;
                  IF OPTEXT = PLUS THEN SIGN:=+1;
                  IF SIGN <> 0 THEN INSERT(48,OPTEXT,0);
                  DECBIN(OPTEXT,TOM);
                  IF PAR < 0 THEN
                  BEGIN IF SIGN=0 THEN GOTO 1020; PAR:=-2;
                      IF SIGN=1 THEN PAR:=-1; GOTO 1050;
                  END;
                  IF SIGN=0 THEN
                  BEGIN SIGN:=1; PAR:=0; END;
                  PAR:=PAR+TOM*SIGN;
                  IF PAR<0 THEN GOTO 1020;
1050:             INSERT(PAR SHIFT(-8),OPDUMMY,0);
                  INSERT(PAR, OPDUMMY,1);
                  MOVE(OPDUMMY,0,OPDUMMY,Q*2,2);
                  IF OPTEST <> 0 THEN GOTO 1040;
                  GOTO 1020;
1060:             IF OPTEXT=STATE THEN IF Q<NOQ THEN GOTO 1015;
              END UNTIL Q>=NOQ; GOTO 1000;
1070:         OPMESS(RUNTXT);
          END;
```

```
PROCEDURE OPSTOP;
BEGIN
    OPWAIT(LENGTH);
    OPTEXT:=OPSTRING;
    OPIN(OPSTRING);
    IF OPTEXT=CONT THEN CONTINUE;
    IF OPTEXT=STOP THEN GOTO 1;
    IF OPTEXT = SUSPEND THEN SAVEDSUSPEND:= 1;
END;

PROCEDURE SHOWERROR;
BEGIN
    ERRORNO:=20;
    WHILE MASK>0 DO
    BEGIN
        MASK:=MASK SHIFT 1;
        ERRORNO:=ERRORNO+1
    END;
    BINDEC(ERRORNO,OPTEXT);
    OPMESS(OPTEXT); OPMESS(ENDLINE);
END;

PROCEDURE CRERROR;
BEGIN
    IF IN.Z0 AND 2'10000 <> 0 THEN
    BEGIN
        SAVEDSUSPEND:= -SAVEDSUSPEND;
        GOTO 9;
    END;
    IF CARDSREAD=0 THEN GOTO 9;
    IF IN.Z0<>0 THEN
    BEGIN
        OPMESS(CRTXT);
        MASK:=IN.Z0;
        SHOWERROR;
    END;
    REPEAT OPSTOP UNTIL OPTEXT=START;
    OPMESS(RUNTXT);
    IF IN.Z0 AND 8'040000 <>0 THEN !CARD READER OFFLINE!
    REPEATSHARE(IN);
END;

PROCEDURE MTOUTERROR;
BEGIN
    IF OUT.Z0 AND 8'043000 = 0 THEN BLOCKNO:=OUT.ZBLOCK;
    NEXTMT:= OUT.Z0 AND 8'000020;
    OUT.Z0:=OUT.Z0 - NEXTMT;
    IF OUT.Z0 SHIFT 1 < 0 THEN OPMESS(MTMOUNTTAPE);
    IF OUT.Z0 SHIFT 1 >=0 THEN
    IF OUT.Z0 <> 0 THEN
    BEGIN
        OPMESS(MTTXT);
        MASK:=OUT.Z0;
        SHOWERROR;
    END;
    IF OUT.Z0<>0 THEN
    BEGIN
        REPEAT OPSTOP UNTIL OPTEXT=START;
        OPMESS(RUNTXT);
        IF OUT.Z0 AND 8'063352 <> 0 THEN
        REPEATSHARE(OUT);
    END;
END;
```

```
          PROCEDURE CHANGETAPE;
          BEGIN
              OPMESS(ENDTAPE);
99:           OPWAIT(LENGTH);
              OPTEXT:=OPSTRING;
              OPIN(OPSTRING);
              IF OPTEXT=CONT THEN
              BEGIN
                  CONTINUE;
                  GOTO 99;
              END;
              IF OPTEXT=STOP THEN
              BEGIN
                  NEXTMT:=0;
                  CLOSE(OUT,1);
                  OPMESS(MTMOUNTTAPE);
                  FILENO:=1;
                  BLOCKNO:=1;
                  REPEAT OPSTOP UNTIL OPTEXT=START;
                  OPEN(OUT,3);
                  SETPOSITION(OUT,1,1);
              END;
          END;
```

```
        BEGIN
        BLOCKNO:=1; FILENO:=1; REWIND:=-1; NEXTMT:=0;
        FIXRECS:=-1; MAXCOL:=80; MINCOL:=80; BLOCKED:=25;
        OPCONT:=FIFTEEN; NEXTCONT:=FIVE; WBLOCKED:=0; CARDSREAD:=0;
        OUT.ZFILE:=1; OUT.ZBLOCK:=1; OPIN(OPSTRING); SAVEDSUSPEND:= 0;

1:      OPCOM; INITPOSITION;

2:      IF OPTEST<>0 THEN OPSTOP;

3:      GETREC(IN,INLENGTH);
        IF FIXRECS=-1 THEN
        BEGIN
            INLENGTH:=MAXCOL; OUTLENGTH:=MAXCOL;
            GOTO 4;
        END;
        IF INLENGTH<=MINCOL THEN OUTLENGTH:=MINCOL;
        IF INLENGTH>MINCOL THEN
        BEGIN
            OUTLENGTH:=INLENGTH;
            IF OUTLENGTH>MAXCOL THEN OUTLENGTH:=MAXCOL;
        END;

4:      PUTREC(OUT,OUTLENGTH); CARDSREAD:=CARDSREAD+1;
        MOVE(IN|,0,OUT|,0,INLENGTH); IF INLENGTH<OUTLENGTH THEN
        MOVE(SPACES,0,OUT|,INLENGTH,OUTLENGTH-INLENGTH);
        IF BLOCKNO<>OUT.ZBLOCK THEN
        BEGIN
            BLOCKNO:=OUT.ZBLOCK;
            WBLOCKED:=0;
            IF NEXTMT<>0 THEN CHANGETAPE;
            GOTO 2;
        END;
        WBLOCKED:=WBLOCKED+1;
        IF WBLOCKED<BLOCKED THEN GOTO 3;
        OUTBLOCK(OUT);
        BLOCKNO:=OUT.ZBLOCK;
        WBLOCKED:=0;
        IF NEXTMT<>0 THEN CHANGETAPE;
        GOTO 2;

9:      IF SAVEDSUSPEND = -1 THEN
        BEGIN
            OUTBLOCK(OUT);
            BLOCKNO:= OUT.ZBLOCK;
            CLOSE(IN,1);
            CLOSE(OUT,1);
            CARDSREAD:= 0;
            WBLOCKED:= 0;
            SAVEDSUSPEND:= 0;
            OPMESS(SUSTXT);
            GOTO 12;
        END;
        IF CARDSREAD=0 THEN GOTO 10;
        IF OPCONT = FIVE THEN GOTO 10;
        GOTO 11;
10:     OPMESS(CRMOUNTDECK);
        GOTO 12;
11:     CLOSE(IN,1);
        CLOSE(OUT,REWIND+2);
        BLOCKNO:=1; FILENO:=FILENO+1;
        IF REWIND=-1 THEN FILENO:=1;
        IF FILENO=1 THEN NEXTMT:=0; CARDSREAD:=0; WBLOCKED:=0;
        OPMESS(EOJTXT);
12:     REPEAT OPSTOP UNTIL OPTEXT=START;
        INITPOSITION; OPMESS(RUNTXT); GOTO 2;
        END;
SIZE: 02783
```

Part Four
Appendix

4.1 The Type Section

A MUSIL program may have a Type Section between the Constant Section and the Variable Section. The purpose of the Type Section is to provide a shorthand for defining types, or categories, of variables. It is most important to remember that the definitions in the Type Section are not substitutes for definitions in the Variable Section. Type Section definitions are used to define a structure of a variable type. For example, if one has two or more file definitions in the Variable Section, and both definitions have the same structure, then one can save some time and effort by defining the structure as a type within the Type Section and referring to this type in the Variable Section.

4.1.1 Section Structure

The Type Section begins with the word

```
TYPE
```

not followed by any punctuation. It ends with a semicolon and its statements are separated from one another by semicolons.

4.1.2 Integer and String Types

The format for integer type definition is

```
name = INTEGER;
```

and for a string variable type, it is

```
name = STRING(n);
```

where n is the string length in bytes, for example,

```
I = INTEGER;
LINE = STRING(132);
```

Once such definitions have been set up in the Type Section, we may use them to define variables in the Variable Section, thus:

```
VAR
        L,M,N:I;
        D,E:LINE;
```

which defines the variables L, M, and N as INTEGER and D and E as STRING(132).

As can easily be seen, the Type Section is not terribly useful in defining integer or string variables. It is much more useful, however, in defining file and record variables.

4.1.3 Record and File Types

Record and file types are defined in the Type Section in a way very reminiscent of the definition of records and files in the Variable Section. The only difference is that the colon used to assign a name to the record or file is replaced by an equals sign, for example,

```
TYPE
          PLINE = RECORD
                        L1:STRING(20);
                        L2:STRING(50)
                        END;
          IN    = FILE
                        'MT0',14,1,600,FB
                        OF PLINE;
```

Once these definitions have been made of a record and a file structure, then in the Variable Section we can write, for example,

```
VAR
          IN1,IN2:IN;
          LINE1,LINE2,LINE3:PLINE;
```

which defines the file variables IN1 and IN2 as having the same structure as file type IN, and the record variables LINE1, LINE2, and LINE3 as having the structure of type PLINE.

4.1.4 Possible Errors

In general one may make the same errors in the Type Section as one can make in the Variable Section. One can also forget to use the equals sign properly. But the most frequent error is to forget to define the appropriate variables in the Variable Section and to try to use type definitions as variable definitions.

4.2 Reference List of MUSIL Commands, Operators, and Symbols

4.2.1 MUSIL Commands

COMMAND	DESCRIPTION
BINDEC(binary value name, decimal value name)	Convert binary to decimal
CLOSE(filename,release)	Close file
CONVERT(stringname,stringname,tablename,length)	Code conversion
DECBIN(decimal value name,binary value name)	Convert decimal to binary
GETREC(filename,variable name)	Get next record for processing
GOTO label	Unconditional branching
IF relation THEN statement	Conditional branching
IF relation THEN statement ELSE statement	Conditional branching
INBLOCK(filename)	Prepare a block for input
INCHAR(filename,integer variable name)	Get next character for processing
INSERT(integer value or name,string variable name, integer value or name)	Insert byte value in string at place designated
MOVE(string name,from n+1th byte,to string name,from n+1th byte,for number of bytes)	Move bytes from one string to another
OPEN(filename,mode)	Open file
OPIN(string variable name)	Input string from console
OPMESS(string variable name)	Output string to console
OPSTATUS(filename.Z0,string constant)	Display status word
OPTEST	Test for successful input from console
OPWAIT(integer variable name)	Wait for operator input
OUTBLOCK(filename)	Prepare a block for output
OUTCHAR(filename,constant)	Prepare a character for output
OUTTEXT(filename,stringname)	Insert text into output file
PUTREC(filename,integer value or name)	Prepare next record for output
REPEAT statement UNTIL relation	Repeat command
REPEATSHARE(filename)	Restart command for a GIVEUP procedure
SETPOSITION(filename,filenumber,block number)	Position medium
TRANSFER(filename,length,mode)	Output or input data
WAITTRANSFER(filename)	Wait for completion of output or input
WAITZONE(filename)	Pause
WHILE relation DO statement	Repeat command
$COPY	Copy code from a second source and place it here
$END	Stop copying from this source
PROCEDURE name (parameters) CODEBODY external identification;	Define a code procedure

4.2.2 MUSIL Operators and Symbols

! text !	Comment
#list#	Table of numbers
<number>	Byte value
"text"	Text string of ASCII characters or byte values
,	Constant section separator
:	Variable definition
=	Constant declaration, Type definition
:=	Value assignment
;	Statement separator
BYTE	Byte value
WORD	Word value
nnnnn:	Label
BEGIN ... END	Compound statement
( )	Parentheses
+	Addition
-	Subtraction
*	Multiplication
/	Division
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal to
<>	Not equal to
m AND n	Logical AND
m SHIFT n	Shift m left n bytes
m SHIFT (-n)	Shift m right n bytes
m EXTRACT n	Extract n bits of m
U	Undefined and unblocked
UB	Undefined and blocked
F	Fixed and unblocked
FB	Fixed and blocked
V	Variable and unblocked (IBM V format)
VB	Variable and blocked (IBM VB format)

4.3 ASCII Code Table

Decimal Representation	7-Bit Octal Code	Character	Decimal Representation	7-Bit Octal Code	Character	Decimal Representation	7-Bit Octal Code	Character
0	000	NUL	43	053	+	86	126	V
1	001	SOH	44	054	,	87	127	W
2	002	STX	45	055	-	88	130	X
3	003	ETX	46	056	.	89	131	Y
4	004	EOT	47	057	/	90	132	Z
5	005	ENQ	48	060	0	91	133	
6	006	ACK	49	061	1	92	134	***
7	007	BEL	50	062	2	93	135	
8	010	BS	51	063	3	94	136	↑
9	011	HT	52	064	4	95	137	←
10	012	LF	53	065	5	96	140	·
11	013	VT	54	066	6	97	141	a
12	014	FF	55	067	7	98	142	b
13	015	CR	56	070	8	99	143	c
14	016	SO	57	071	9	100	144	d
** 15	017	SI	58	072	:	101	145	e
16	020	DLE	59	073	;	102	146	f
17	021	DCI	60	074	<	103	147	g
18	022	DC2	61	075	=	104	150	h
19	023	DC3	62	076	>	105	151	i
20	024	DC4	63	077	?	106	152	j
21	025	NAK	64	100	@	107	153	k
22	026	SYN	65	101	A	108	154	l
23	027	ETB	66	102	B	109	155	m
24	030	CAN	67	103	C	110	156	n
25	031	EM	68	104	D	111	157	o
26	032	SUB	68	105	E	112	160	p
27	033	ESC	70	106	F	113	161	q
28	034	FS	71	107	G	114	162	r
29	035	GS	72	110	H	115	163	s
30	036	RS	73	111	I	116	164	t
31	037	US	74	112	J	117	165	u
32	040	SP	75	113	K	118	166	v
33	041	!	76	114	L	119	167	w
34	042	"	77	115	M	129	170	x
35	043	#	78	116	N	121	171	y
36	044	$	79	117	O	122	172	z
37	045	%	80	120	P	123	173	
38	046	&	81	121	Q	124	174	***
39	047	'	82	122	R	125	175	
40	050	(	83	123	S	126	176	~
41	051	)	84	124	T	127	177	DEL
42	052	*	85	125	U			

** Special control characters.
Will be interpreted in accordance with actual device specifications.

*** Reserved for national characters.

4.4 Device Reference Tables

Device error numbers are explained in the RC 3600 Operator's Manual and on the RC 3600 Data Conversion System Operator's Reference Card. Device information of use to the MUSIL programmer is as follows:

4.4.1 Kind Table

bit 15	set if device is character-oriented
14	set if full blocks should be transferred
13	set if positioning has any effect
12	set if an operation may be repeated
11	set if the device is a cataloged disc file

Examples:

14 =	1110	Magnetic tape station
1 =	0001	Line printer
3 =	0011	Line printer
2 =	0010	Card reader
1 =	0001	Teletype
1 =	0001	Paper tape punch
1 =	0001	Paper tape reader

4.4.2 Operation Mode Table

Paper tape reader driver - PTR

1	binary, the input character is delivered
5	odd parity, the most significant bit is removed
9	even parity, the most significant bit is removed

Paper tape punch driver - PTP

3	binary, the converted character is output
7	odd parity, the converted character is augmented by the complement of its parity in the most significant position
11	even parity

Line printer driver - LPn

3	the converted characters are output
7	the first byte of output is interpreted as a carriage control word

Magnetic tape driver - MTn

1	read packed, byte limit = 18
5	read packed, byte limit = 0
9	read unpacked, byte limit = 18
13	read unpacked, byte limit = 0
3	write

Concerning the Magnetic tape driver: when using 7 track tape, if 4096 is added to any of the operation modes, then the reading or writing will be done in even parity. If the number 8192 is added to any of the mode numbers, then the resulting number will cause reading or writing to be done in the tape's lower density.

For example, 4099 signifies write with even parity, for 4099 = 3 +4096.

Card reader driver - CDR

1	read binary byte
5	read decimal punched cards
21	read decimal punched cards
33	read decimal punched cards and skip trailing blank columns (a minimum of ten columns are read)

Card reader punch driver - RDP

1	read binary bytes
5	read decimal word
9	read binary word
21	read decimal bytes, skip trailing blanks (a minimum of ten columns is read)
11	punch decimal byte
19	print decimal byte
27	punch and print decimal byte
47	punch binary word

If 256 is added to any of the read mode numbers then the resultant sum used as an operation mode causes Hopper #2 to be selected. Adding 64 causes the card to be released and a new card to be fed.

Adding 256 to any of the punch mode numbers causes Stacker #2 to be selected. Adding 64 causes a card to be fed before the operation is performed.

For example, 257 = 1 + 256 means Read binary bytes from a card in the second hopper.

Plotter driver - PLT

3	write byte

Flexible disc driver - FDn

1	read
3	write
5	read, non-skip
7	write and check read

Charaband printer driver - CP0

3	print converted characters
7	interpret first byte as carriage control
15	output to load Direct Access Vertical Format Unit at 6 lpi
31	output to the DAVF unit at 8 lpi

Cassette tape unit driver - CTn

CT0

1	read one block per ECMA 34 version 2
3	write one block per ECMA 34 version 2
+4	read data blocks continuously
9	read one block per ECMA 34 version 1
17	read one block without checking

CT4

1	read one block per ECMA 34 version 2
3	write per ECMA 34 version 2
5	continuous read per ECMA 34 version 2
7	write with control read per ECMA 34 version 2
9	read one block per ECMA 34 version 1
11	write per ECMA 34 version 1
13	continuous read per ECMA 34 version 1
15	write with control read per ECMA 34 version 1
17	read one block without check
21	continuous read without check

Serial printer driver - SPT

3 write converted character

7 interpret first byte as carriage control

Cartridge disc driver - DKPn

1 read

3 write

Communications equipment

See terminal reference cards.

NB: The Operation Code, which represents whether the operation is an output or an input operation, consists of the two least significant bits of the Operation Mode.

4.5 Program Production

Once one has written a MUSIL program down on paper, one may enter it onto some medium, such as cards or tape, or one may key it directly into the machine. For the latter choice, it is convenient to use MUSIL Text Editor, which takes in the MUSIL source code as data to itself. Text Editor can also be used to change programs that did not compile properly or to up-date job programs.

Instructions for using MUSIL Text Editor can be found in the manual MUSIL Text Editor Version Two. A list of Text Editor commands and error messages can also be found on the RC 3600 MUSIL Programmer's Reference Card.

4.5.1 Compilation

Once the MUSIL source code has been written and debugged, it must be compiled into MUSIL object code for execution by the MUSIL Interpreter. On the RC 3600 MUSIL Programmer's Reference Card can be found the procedures and commands for compilation.

4.5.2 MUSIL Compiler Error Messages

During compilation errors give rise to the following numbers on the listing or on the operator's console:

020202	Number overflow, a numeric constant exceeds 65535, or 16 bits.
020301	Illegal character in input.
030102	< appearing within a string is not followed by a numeric literal.
030202	The construct < number is not followed by a >.
030302	The number between < and > exceeds an 8-bit byte value.
030403	Core overflow, produced code exceeds available space.
030503	Core overflow, code contains too many relocation bits.
040105	Name conflict in Constant Section.
040205	Name conflict in Type Section.
040302	Syntax in Type Section, no = following an identifier.
040405	Name conflict in Variable Section.
040506	File variable with 0 buffers.
040602	Procedure head not followed by ,
040702	Procedure without legal identifier or with name conflict.
050102	Type is no identifier.
050202	(is missing after string.
050302	Length undefined for string.
050402	String with length 255 declared.
050502	) is missing after string.
050604	Undefined type identifier. Note that no forward declarations are allowed.
050702	Improper termination of type specification.
051002	Field of type different from string.
051102	Incorrect use of FROM.
051205	Name conflict in GIVEUP procedure.
051304	Conversion table undeclared.
051406	Conversion table type error.
060206	Double defined label.
060302	Variable is no identifier. Or undeclared.
060402	. is not followed by identifier or by undeclared field.
060504	Identifier undeclared.
060606	Type error with BYTE or WORD.
060702	Relational operator missing.
061002	Procedure statement with missing)

061102	Type error in procedure parameter.
061306	Illegal number of parameters.
061406	Type error with operator.
061506	Overflow of work registers. Expression too complex.

Error Messages which cause skipping of program parts:

000040	Syntax in section delimiter.
000041	Syntax in constant declaration.
000042	Syntax in table declaration.
000043	Type specification incorrectly terminated.
000044	Variable declaration incorrectly terminated.
000045	
000046	
000051	Syntax in field list.
000052	Syntax in file declaration.
000063	Incomprehensible statement.
000064	Incorrect label declaration.
000065	Incomprehensible expression.

4.5.3 Copying MUSIL Program Parts

At the time of compilation parts of one MUSIL source may be copied into a place within another MUSIL source. The sources must, of course, not be located on the same device. To do the copying the command

$COPY

is inserted in the program into which the code is to be copied at the location where the code copied is to be placed. The command

$END

is inserted into the source whose code is being copied and is placed after the last instruction to be copied. Copying begins from the first statement of the copied program.

4.6 List of Reserved MUSIL Words

To produce a tape or card deck or disc with a full run, including systems software and one or more applications programs in MUSIL object code, one must use the program generator supplied as part of one's program production package.

4.6 List of reserved MUSIL words

AND	GOTO	RECORD
		REMOVEENTRY
BEGIN	IF	REPEAT
BINDEC	INBLOCK	REPEATSHARE
BYTE	INCHAR	
	INITCAT	SETPOSITION
CHANGEENTRY	INSERT	SHIFT
CLOSE	INTEGER	STRING
CODEBODY		
CONV	LOOKUPENTRY	THEN
CONVERT		TRANSFER
CONST	MOVE	TRANSLATE
CREATEENTRY		TYPE
	OF	
DECBIN	OPEN	UNTIL
DO	OPIN	
	OPMESS	VAR
ELSE	OPSTATUS	
END	OPTEST	WAITTRANSFER
EXTRACT	OPWAIT	WAITZONE
	OUTBLOCK	WHILE
FILE	OUTCHAR	WORD
FROM	OUTTEXT	
GETREC	PROCEDURE	
GIVEUP	PUTREC	

READER'S COMMENTS

MUSIL Programming Guide
RCSL 42 - i 0344

A/S Regnecentralen maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback - your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position:

Name:

Organization:

Address:

Department:

Date:

Thank you!

RETURN LETTER - CONTENTS AND LAYOUT

Cut along the dotted line

Fold here

Do not tear - Fold here and staple

Affix
postage
here

A/S REGNECENTRALEN
Marketing Department
Falkoner Allé 1
2000 Copenhagen F
Denmark

A/S REGNECENTRALEN

HEADQUARTERS: FALKONER ALLE 1; DK-2000 COPENHAGEN F · DENMARK
Phone: (01) 10 53 66 · Telex: 16282 rc hq dk · Cables: regnecentralen

INTERNATIONAL

EASTERN EUROPE
A/S REGNECENTRALEN
Glostrup, Denmark, (02) 96 53 66

SUBSIDIARIES

AUSTRIA
RC – SCANIPS COMPUTER
HANDELSGESELLSCHAFT mbH
Vienna, (0222) 36 21 41

FINLAND
OY RC – SCANIPS AB
Helsinki, (90) 31 64 00

HOLLAND
REGNECENTRALEN (NEDERLAND) B.V.
Rotterdam, (010) 21 62 44

NORWAY
A/S RC – SCANIPS
Oslo, (02) 35 75 80

SWEDEN
RC – SCANIPS AB
Stockholm, (08) 34 91 55

SWITZERLAND
RC – SCANIPS (SCHWEIZ) AG
Basel, (061) 22 90 71

UNITED KINGDOM
REGNECENTRALEN LTD.
London, (01) 439 9346

WEST GERMANY
RC – GIER ELECTRONICS G.m.b.H.
Hannover, (0511) 6 79 71

REPRESENTATIVES

FRANCE
SORED S.a.r.l.
Nanterre, (1) 204 2800

HUNGARY
HUNGAGENT AG
Budapest, 88 61 80

TECHNICAL ADVISORY REPRESENTATIVES

POLAND
ZETO
Wroclaw, 45 431

RUMANIA
I.I.R.U.C.
Bucharest, 33 21 57

HUNGARY
NOTO-OSZV
Budapest, 66 84 11