introduction to musil

rc 3600

Introduction

MUSIL is a programming language that was designed for the specific purpose of facilitating input/output information processing. Therefore, it is primarily concerned with data in its aspect as text, rather than in its aspect as numerical values. MUSIL is, thus, not designed for computational purposes.

MUSIL is, secondly, designed to facilitate communications between the programmed operation and the machine operator. This means that it is designed to satisfy realtime programming needs.

Finally, MUSIL is suitable for data communications. Its instruction set includes a considerable repertory of error-handling instructions.

The purpose of this booklet is to introduce the reader to MUSIL programming. It assumes that the reader is already familiar with some other programming language, whether a language of the assembly type or of the higher level type. The booklet does not, therefore, describe all the possible MUSIL instructions, and it does not discuss the interaction of MUSIL with the underlying MUS operational system. For advanced MUSIL programming, the reader should refer to the MUSIL reference manual and to the MUS manuals.

There is a text editor program that can be used by the programmer to change or correct his programs sitting at the console device of the RC 3600 machine. The use of this program for such purposes is described in the RC 3600 MUSIL Text Editor Manual.

About Musil

MUSIL is a language that can operate both on whole files and on individual characters within files. Thus, it shares some of the characteristics of assembly languages and also of higher-lever languages. The programmer with previons experience with I/0 programming in an assembler language should be able to learn MUSIL in a day or two. The programmer whose previous experience has been with languages such as FORTRAN or ALGOL may find it advisable initially to write programs that handle whole files, and then progress to the full set of MUSIL instructions.

Whatever the level of knowledge of the programmer, certain programming conventions should be followed when using MUSIL, so that programmers other than a program's original author will find it easy to understand, up-date, modify, and/or correct the original program. This is a very important point to remember, as it is estimated that up to fifty percent of programmer time at any programming installation is occupied with work on old programs. Following correct programming conventions will also help you to write error-free programs more quickly.

The first principle of good programming technique is that programs should be written in modular fashion. MUSIL provides facilities to help you follow this principle easily. A MUSIL program is written in sections. The first section is the *constant section*. This is followed by the *type section*, the *variable section*, and the main section. Within the main section *procedures* are first defined. In the final part of the main section there should be -- as far as is practical -- only calls to procedures. Programs written this way are easy to read, modify, and document.

The second principle of good programming technique is full, clear, and adequate documentation. Each program should begin with a comment section which describes the purpose and operation of the program. Each procedure should be proceded by a comment section that describes the purpose of the procedure and the conditions under which it will be called. Each line of the main program (and of very long procedures too) should be explained by comments.

In MUSIL comments are written within two exclamation signs:

! THIS IS A COMMENT !

Thus the overall structure of a MUSIL program should ressemble this model:

```
! comments describing program
     •
     •
     •                               !
 constant section
 type section
 variable section

 ! description of first procedure !
 first procedure
 ! description of second procedure !
 second procedure
                       •
                       •
                       •
 ! description of last procedure !
 last procedure
 main program    ! comments to main program !
      •                      •
      •                      •
      •                      •
```

The third principle of good programming practice is readability. That is, many different people may have to read your program, or you may have to read it long after you have written it (and forgotten it). Thus, it is advisable to write the main program in such a way that only one instruction, or two closely related instructions, appears on each line, with the remainder of the line being used for comments. Longer procedures should be written in this way also.

Modularity, documentation, and readability will not only make your programs more usefull. They will also help you to write better programs faster, and they will allow you to achieve a maximum of error-free coding. They are well worth the time they take.

The Constant Section

The first part of a MUSIL program is the Constant Section. In it several different sorts of constants can be defined. The simplest is the definition of a simple numerical value:

```
ALPHA = 45,
```

is such an example. This statement assignes the value decimal 45 to the world ALPHA. ALPHA'S value could have been set in octal or in binary, as well as in decimal, but at any rate, the value of ALPHA must not exceed 16 bits. That is, ALPHA must be a value between decimal –32768 and +32767. The name of the value, in this case **ALPHA**, can be as long as you like. The system will identify it by its first seven characters and the total number of characters in it. The name of the value must, furthermore, begin with a letter and include no symbols other than letters and numbers. Notice that each assignment in the Constant Section must close with a comma, and that the system will ignore spaces.

Some other examples of numerical values might be:

```
NUM123 = 2'011001,     ! a binary number !
NUM555 = –8'775,       ! an octal number !
ACT88B = +23005,       ! a decimal number !
```

Decimal points *cannot* be used.

The Constant Section begins with the keyword

```
CONST
```

not followed by any puctuation. As stated above, every definition is followed by a comma:

```
ALPHA = 45,
BETA1 = 67,
```

and the last entry in the section is followed by a semicolon:

```
CONST
      ALPHA = 45,
      BETA1 = –8'377,
      GAMMA = +2'0011;
```

Though the system will ignore spaces that occur between parts of a statement, blanks must not occur within the name of the value or within the numerical value itself. The following statements are *not* allowed:

```
GA MMA = +2'0011,      ! error in name !
BETA   = –     8'377,  ! error after sign !
PHI   = –2'00 11,      ! error in value !
```

Besides integers, other sorts of constants can be defined in this section. The most common one is the string of characters representing an *ascii* text. For example,

```
ALPHA2 = 'THIS IS ALPHA2',
```

which gives the name ALPHA2 to the text THIS IS ALPHA2. Such a text cannot, obviously, be operated on numerically, but it can later on be assigned to a variable as its current value. It can also be used in text comparisons. And it can be output on the operator's console.

Strings can be enclosed within either single or double quotes, and no error occurs if the single and double quotes are mixed. Thus, it is all right to write

```
ALPHA3 = "THIS IS ALPHAS SECOND VALUE",
ALPHA4 = "THIS IS OK',
ALPHA5 = 'THIS IS OK TOO",
```

String constants defined in this section are stored in their locations left-justified and with a binary zero at the end of the text. When they are read out to another location, or to an output device, the binary zero is stripped off. Therefore, it is important to remember that this terminal zero will not be carried with the text when it is later on assigned to a variable *and then* output to the console. The absence of this binary zero will cause the console device to keep on printing after the output text has been completed. To avoid such a situation, you should place a binary zero after each text that will be assigned to a variable *and then* output from that variable. This is done in the following way:

```
ALPH   = 'THIS WILL BE OUTPUT <0>',
```

Strange things can happen if the above method is not employed. Say, I have in ALPHA20 the text THIS MESSAGE IS WRONG. If somewhere in my program, I move into ALPHA20 the text THIS IS ALPHA, then on outputting ALPHA20 I would get

```
THIS IS ALPHAIS WRONG
```

The use of the final zero will eliminate such situations. Of course, string constants that will not be moved around in the program need not have the binary zero put after them, for the compiler will do this automatically in the execution of the Constant Section.

Text strings may be defined for strings of ASCII values. If we write

```
ALPHA = '<45>',
```

we have a text string, and though we cannot perform arithmetic on it, it can be assigned to a variable, compared with another text string, or output to a device. For this sort of statement the binary value 45 goes into the location. Since the ASCII code for decimal 45 is a minus sign, a minus sign is put into ALPHA. If we write

```
BETA = '<8' 26>',
```

then the ASCII code for V goes into BETA (left-justified). Similarly, we can define ASCII representations for carriage return, end of text, or whatever. *This is the only way to include control characters in a text.*

Using this method any symbol can be output, including "Bell", ', and ".

An ASCII code table follows for your reference.

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	DC1	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	69	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	120	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			

You can also write strings of ASCII characters:

```
ALPHA31 = '<45><0><10>',
```

which sets into ALPHA31 the string meaning

minus sign NUL Line feed

One can also define tables of constants in the Constant Section. These constants are also text, that is string constants, and the items in the table cannot be operated on arithmetically. A constant table might be set up in this way:

```
LPTABLE = #14 0 64 89 56 8'377 0 65#,
```

Notice that the punctuation used between the elements of a table is the blank. Note also the two following statements are equivalent

```
ALPHA = #45#,   and   ALPHA = '<45>',
```

The numerical sign is a shorthand notation that allows the programmer to avoid cumbersome forms such as

```
LPTABLE = '<14><0><64><89><56><8'377><0>
65 ',
```

Finally, constants useful in error routines can be defined, for example,

```
STATUS = 'DISCONNECTED<10><0>
          OFFLINE<10><0><0  ><0>
          <0> <0><0><0><0>EOF
          <10><0> B8<10><0> PARITY
          <10><0> EM <10><0><0><0>
          <0><0>',
```

which will, when used with certain instructions, display the appropriate messages on the operator's console.

The Type Section

The second section that might appear in a *musil* program is the Type Section, but this is not, strictly speaking, a necessary part of a MUSIL program. The Type Section in fact is only a place where a kind of shorthand notation is provided for defining variable types, or categories, so that several variables that have the same structure can later on be defined more easily in the third section of the program, the Variable Section. There this is done by referring to the type definition that applies to all of them. In this section, then, variables are not defined, but categories of variables are defined for later reference in the next section.

Variable types are defined in the Type Section by identifying them, e.g., by specifying that they are to be integers, files, or records, etc., by associating them with an identifier and by describing their structure, if any. In the last case an example might be a situation in which we describe the structure of a file by saying how many records it contains and what the records look like. Many examples will be found below.

The Type Section begins with the word

```
TYPE
```

not followed by any punctuation.

We may define scalar types. These may be integers or strings. Such an integer type definition might look like this:

```
I = INTEGER;
```

which sets up a category, called I, whose members will all be 16-bit signed binary integers.

Each statement in the Type Section is terminated by a semicolon.

We may define string types here, viz.,

```
LINE = STRING (20);
```

This defines the category called LINE and specifies that it shall have as members strings consisting of twenty 8-bit bytes.

Besides scalar types, we may also define record types here.

```
TYPE
    PLINE = RECORD
              L1: STRING (20);
              L2: STRING (15);
              L3: STRING (45)
              END;
```

defines a record type, to be called PLINE, which consists of strings of 20, 15, and 45 bytes in sequence, and called respectively, L1, L2, and L3. We might have written this definition in an equivalent way:

```
TYPE
    LINE1 = STRING (20);
    LINE2 = STRING (15);
    LINE3 = STRING (45)
    PLINE = RECORD
              L1: LINE1;
              L2: LINE2;
              L3: LINE3
              END;
```

or we might have used a mixture of the two equivalent forms:

```
PLINE = RECORD
          L1: LINE1;
          L2: STRING (15);
          L3: STRING (45)
          END;
```

Note that *punctuation cannot come before an END.*

Such record definitions are useful in situations in which control characters will be used.

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

sets up a record type definition for a record whose first two characters are text strings of one character each and called, respectively, CCW and TEST. These are followed by a string of 132 characters. We furthermore define a name for the first two characters taken together. We call them STOPF. One may also write

```
CCW, TEST: S;
```

Finally, we can define file types in the Type Section. The coding

```
IN = FILE
     'MTO', 14, 1, 600, FB
     OF PLINE;
```

sets up a file of records with the record structure previously defined when we described PLINE above. (In this coding we might have replaced PLINE by its definition.) The coding further tells us that the device is call MTO. *This name must have been defined in the device's driver program.*

One is permitted to use single (' ') or double (" ") quotes around the device name, which can be up to six characters in length.

Following the device name, appears the decimal representation for the binary code that tells the central unit what to expect from the device and its operation. At present the following kind bits are defined:

bit 15	char	is set if the device transfers information character-by-character;
bit 14	blocked	is set if full blocks are transfered as units;
bit 13	positionable	is set if positioning is effectual on the device;
bit 12	repeatable	is set if an operation can be repeated.

For example, binary 1110 equals decimal 14, so that our MTO is not character-oriented, but is block-oriented and positionable, and can also repeat I/0 operations.

Further examples might be

0001	line printer	0001	teletype
0011	line printer	0001	paper tape punch
0010	card reader	0001	paper tape reader

In our example, MTO is, obviously, a magnetic tape station.

Following the kind definition, we find the decimal representation for the number of buffers. The maximum that can be used is 64. One determines the number of buffers for the device by trading off execution time against space in core. You will probably want to try a number of possibilities for each of the programs you write, as the determination of this number can influence severely the speed with which your programmed operation proceeds.

Our example, then, uses one buffer for the device MTO.

Next comes the blocklength in number of bytes. In our example there are 600 characters per block. This number is limited by core size.

The next slot is filled by a character or by two characters. These represent the record format. The possibilities for this slot are

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

That takes care of our example, but two more file type definers are possible.

Consider the example

```
LPT = FILE
      'LPT', 1, 2, 50, U;
      GIVEUP LPTERRORS, 2'1100001111111111;
      CONV LPTTABLE
      OF STRING (50);
```

Files of the type called LPT, then, operate on the device named LPT. (As defined in the device's driver program.) Two buffers are set up for it. Block length is 50 characters, and it is unblocked. Furthermore, there will be a procedure, called LPTERRORS and defined later on in the program, that will specify some action to be taken by the operator and/or the machine if there is an error, an end of file, or any other special situation. What the machine will do is defined by the binary mask, which is described in the device's driver program.

If there is a conversion table related to the file, then it is called here LPTTABLE, and it was defined previously in the constant section, or it will be defined in the Variable Section.

If we had previously defined something like

```
ZLINE = STRING (50);
```

then the last line of the LPT definition could have been written

```
OF ZLINE;
```

Note that OF is not preceeded by punctuation.

Conversion is provided for in the Type (and in the Variable) Section, because doing the conversions outside the main program saves execution time and programmer's time.

When conversion is done, it procedes in the following way: Say we have paper tape input and line printer output. Then if we did no conversion, then whenever an ASCII character came in that was unknown to the line printer, it would be printed out as a space. For example, if a lower case letter was read in, then it would be printed out as a space. If we have a conversion table, then when the machine receives a character, it looks it up in the table and outputs the character it finds there.

Example. If a lower case *a* is read in. This symbol has the ASCII representation decimal 97. Therefore, the driver program looks for the 97th entry in the table. It prints what it finds in this location. Say that the 97th entry in the table was 65. This is the ASCII decimal representation for capital A. Therefore, a capital A is printed out.

Suppose in the Constant Section we had had

0th	LPTTABLE = #	0	0	0	0	0	0	0	0	0	0	10	0	12	13	0	0	
16th		0	0	0	0	0	0	0	0	0	32	0	0	0	0	0	0	0
33rd		33	34	35	36	37	38	39	40	41	42	43	44					
45th		45	46	47	48	49	50	51	52	53	54	55	56	57	58	59		
60th		60	61	62	63	64	65	66	67	68	69	70	71	72	73	74		
75th		75	76	77	78	79	80	81	82	83	84	85	86	87	88	89		
90th		90	91	92	93	94	95	96	65	66	67	68	69	70	71	72	73	
106th		74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	
122nd		90	00	00	00	00	00	00	00	#,								

Refer now to your ASCII table. Comparing the ASCII table with this conversion table, we can see that we must not ignore the character for zero in any case. If a NUL is input, then we look it up in the first place in the table. That is, *in counting table entries, start with zero.*

Continuing, we have the following examples for our table:

input	output
NUL	blank
SOH	blank
.	.
.	.
.	.
.	.
.	.
.	.
.	.
$	$
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
a	A
b	B
.	.
.	.
.	.
.	.
.	.
.	.
.	.
A	A
B	B
.	.
.	.
.	.
.	.
.	.
.	.
m	00

Input for which no entries appear in the conversion table will be output as blanks.

The Type Section interacts with the other sections of the program. For example, if in the Type Section we have

```
I = INTEGER;
```

then later on in the variable section we can make A, B, and C integer variables by setting

```
A, B, C: I;
```

Similarly, we can take a file type defined in the Type Section and use it in a kind of shorthand notation to set up any number of files of similar types. For example, we have defined the file type IN above. If we want to have several files of this type, then in the Variable Section, we might say

```
INFILE1, INFILE2 : IN;
```

This gives the structure of IN to both INFILE1 and INFILE2.

We have defined INREC above. Let us now define

```
OUTREC = RECORD
         CCW: S;
         LINE: STRING (132)
         END;
```

Later on in the main program we can put the contents of the first character of INREC into the first character of OUTREC, for example, *if we have first set up variable of the corresponding types.* We would do this in the Variable Section, for example

```
VAR

         IN: INREC;
         OUT: OUTREC;

main program
         OUT↑.CCW: = IN↑.CCW;
```

Two final cautions: Where in the example above we used the FROM expression, if we have

```
....... FROM n
```

then n cannot be greater than 255.

Also, if you have a line of coding

```
....STRING...FROM...
```

then later on you cannot have something like

```
AB, CD : STRING...FROM...
```

because then you will have defined the string twice.

The Variable Section

The third part of a MUSILprogram is the Variable Section. Here variables are defined and space is set aside for them in core. If the Variable Section has been proceeded by a Type Section, then the process of setting up file variables can be much simplified in the Variable Section. If not, then all the structuring discussed above must be done here, in the Variable Section.

The Variable Section begins with the keyword

VAR

not followed by any punctuation.

We may define integer variables:

```
D: INTEGER;
```

This sets up a location called D, which can accomodate 16-bit signed binary integers.

We may define text string variables:

```
TEXT1 : STRING(20);
```

sets up a location called TEXT1, which can accommodate 20 bytes.

We can define and structure record variables:

```
PRINTLINE:   RECORD
             HEAD: STRING(4);
             TAIL: STRING(4)
             END;
```

which sets up an eight-character record called PRINTLINE in which the first four characters have the name HEAD and the last four have the name TAIL.

We can define and structure a maximum of eight file variables:

```
LPT: FILE
     'LPT', 1, 2, 50, U;
     GIVEUP procedure name, mask
     OF STRING (50);
```

where the meaning of this example was explained in the previous section.

Or records within files can be structured within the file definition:

```
MTO :  FILE
       'MTO', 14, 48, 1000, FB;
       GIVEUP procedure name, mask
       OF RECORD
          COL1: STRING(1);
          COL10: STRING(9)
          END;
```

Some notes on the above examples: After a variable has been defined, you cannot operate on any part of that variable, unless you have given that part a name. Thus, if we have

```
TEXT1: STRING(20);
```

we cannot later perform operations on individual characters within TEXT1.

Similarly, in our example PRINTLINE above, we can operate on the part of PRINTLINE called HEAD, but we cannot operate on parts of HEAD. Similarly, we cannot operate on parts of records of LPT, but we can operate on those parts of MTO called COL1 and COL10.

Secondly, when a block size has been assigned to a file, then output to that file, and assignments to it, must be in blocks of corresponding size. For example, input to, and output from, LPT must be in blocks of 50 characters.

Third, it is most convenient to write mask descriptions in binary, but this is not prerequisite. They may be written in octal or in decimal.

In the Variable Section variables of the same sort can be defined together:

```
D, E, F, G, H: INTEGER;
TEXT1, TEXT2, TEXT3: STRING(40);
```

defines D, E, F, G, and H as integers and TEXT1, TEXT2, and TEXT3 as strings of 40 characters each.

Finally, we had an example in the Type Section of how that section can interact with the Variable Section.

When we later get to the main program, we will want to do certain things with our previously-defined variables. Some of them we might want to do arithmetic with, others we might want to use to compare with the contents of other variables. We will want to make assignments to others. Looking again at the examples given above, we have the following:

Variables defined as INTEGER can be used for arithmetic, comparison, or assignment.
Variables defined as STRING or RECORD can be used for comparison or assignment.
Variables defined as FILE can be used only for I/0 procedures. You *cannot* use them for comparison or assignment (or obviously arithmetic) directly.

Comparison and assignment, with respect to record and file variables, that will be performed in the main program is done with respect to the following facts:

When a file is set up in core, room is reserved for a zone descriptor, which contains I/0 information, for information about operator communications, and for the actual data that will be coming into, and going out of, this location. To refer to any particular part of the data in a file, we use an arrow, thus:

MTO↑.COL1

refers to the current contents of COL1 in MTO. If we have no arrow, we are referring to a part of the zone descriptor, for example

MTO.ZREM

which is something that will be explained later, when we are discussing I/0 procedures.

When structuring records and files, it is possible to give the same name to parts of different records or files. The computer will not get confused, for example, if you refer to

MT0↑.COL1 and CDR↑.COL1

as long as these elements have been previously defined.

The Main Program

The Main Program section is divided into two parts. The first part contains the coding for the various procedures that will be used during program execution. The second part contains the coding that will call these various procedures and inter-relate them with respect to the operation that the program was written to perform. There is no difference between the instruction set that may be used in procedures and the instruction set that may be used in the body of the main program.

In MUSIL procedures are defined first. The structure of a procedure is as follows:

```
PROCEDURE name of procedure;
        BEGIN
          .     ;
          .     ;
          .
         END;
```

Note that every statement except the one before an END is terminated by a semicolon. That is, everything between BEGIN and END is a statement. In fact the entire main program section can be looked at as one compound statement. After a procedure has been defined, it can be referred to by its name, for example by a statement

procedure name;

It can be seen, then, that procedures in MUSIL are the analogy of subroutines in source language programming.

We shall now define the MUSIL instructions that the beginning MUSIL programmer should know.

OPMESS(string variable name)

This instruction outputs the string text contained in the string variable specified in the instruction to the operator's console. That is, it outputs the string text until a <0> is reached. At most 80 bytes will be output, and if there is no final binary zero in the string, then the output will go on for the full 80 bytes anyway, outputting whatever is in core following the text. Of course, the output will be in ASCII text.

OPIN(string variable name)

This instruction allows the operator to input a text string of up to 80 bytes into the string variable specified. This instruction should always be followed by

OPWAIT(LENGTH)

which makes the system wait for the operator input. The number of characters input will be placed automatically into the system-defined variable LENGTH. The use of the instruction OPIN will determine the value of a system-defined function variable

OPTEST

If OPIN has been called and if a text has in fact been input, then this function will take a non-zero value. Otherwise, its value will be zero.

The RC 3600 system operates in binary. Therefore, all input that is not in binary must be converted to binary before it can be operated on arithmetically. Similarly, all output which is not to be in binary must be converted before it is output. The conversion instructions, which follow, should be used close enough to the corresponding I/0 statements to take it easy for the reader of the program to see what is happening.

BINDEC(binary value name, decimal value name);

takes the binary number found in the first variable and puts its decimal value into the second variable of the instruction. The decimal value variable must be previously defined as a string with a minimum of 6 bytes. It will have no sign. If a sign is to be output, then it must be defined separately. The binary value contained in the binary value variable will be converted to 5 decimal digits. The opposite instruction is

DECBIN(decimal variable name,binary variable name);

Here too, the decimal value being converted should have no sign. The decimal value will be converted into a 16-bit binary number. *Note that there is no check for overflow*. The conversion process will stop at the first non-numeric symbol, for example, a plus or minus sign.

If we wish to construct a compound statement, we can do so by using the instruction pair BEGIN and END:

```
BEGIN
  .     ;
  .     ;
  .     ;
  .
END;
```

Note that there is no semicolon after BEGIN or before END.

GOTO label;

This is the ordinary jump instruction found in many programming languages, but in MUSIL certain peculiarities should be observed. If we say, for example, GOTO 31, then there must be a line of code labeled thus:

```
31:  .................
```

Note the colon after the statement label. There are certain logical restrictions on the GOTO statement. You may *not* GOTO a location inside a procedure, if you are not already inside that procedure, but you can GOTO a location in the main program from within a procedure. The use of the GOTO in combination with the BEGIN/END compound statement usage is as follows:

GOTO may be used to jump outside a compound statement, but it may not be used to jump into a compound statement. If the GOTO is used to jump to an END statement, then the END statement must be preceded by a semicolon:

```
          .   ;
          .   ;
          .   ;
        GOTO 60;
          .   ;
          .   ;    ! note semicolon !
    60:END;
```

Assignment instructions move the contents of one location into another location, or move a numerical constant into an appropriate location. In MUSILyou cannot move text strings into a location unless the text string has been defined previously. Thus, you can have

INT1: = 5;

if INT1 was previously defined as an integer variable, but you cannot have

TEXT3: = 'THIS IS THE END';

even if TEXT3 had been previously defined as a string variable. Instead you must in the Constant Section have something like

T3 = 'THIS IS THE END',

and then in the main program you can have

TEXT3: = T3;

You can assign the contents of one location to a location of the same type:

TEXT1: = TEXT2;

but you may not do the following

INT1: = TEXT1; or TEXT1: = INT1;

where INT1 is an integer variable and TEXT1 is a text variable.

You may also not make multiple assignments in one statement. The following are *not* allowed:

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

When text strings are moved, the number of characters that are moved is equal to the minimum of characters in the two values. Thus, if TEXT1 has 10 characters and TEXT2 has 20 characters, then

TEXT1: = TEXT2; or TEXT2: = TEXT1;

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

IF.....THEN.....

MUSIL has the usual IF statement THEN statement construction. For example,

IF TEXT1 = TEXT2 THEN GOTO 35;

The IF may be followed by any relational expression, and the THEN may be followed by any statement, including compound statements. If the relational expression is not true, then the program skips to the next executable statement, and the THEN statement is ignored.

WHILE.....DO....

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

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

If X is never greater than Y, then the DO statement will never be executed.

REPEAT...UNTIL...

The REPEAT statement may be any statement, including compound statements. The UNTIL statement is any relational expression. For example,

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

Note that there is no semicolon after the END. If X is in fact equal to Y when END is reached, then the statement will be executed once.

Relational Symbols. The allowed symbols are

X = Y	The contents of X and Y are equal.
X > Y	The contents of X are greater than the contents of Y. (For texts the comparison is done byte by byte, starting from the left, and the comparison is lexicographic.)
X < Y	The contents of X are smaller than the contents of Y. Comparison of texts is as above.
X < > Y	The contents of X and Y are not the same.
X < = Y	The contents of X are less than or equal to the contents of Y. Comparison as above.
X > = Y	The contents of X are greater than or equal to the contents of Y. Comparison as above.

Arithmetic. **MUSIL** uses these arithmetical operations:

( )	parentheses
+	addition
–	subtraction
*	multiplication
/	division
AND	masking
SHIFT	logical shift left
EXTRACT	bit extraction from the right

MUSIL executes arithmetic operations from left to right, with operations of the same precedence level being executed together. The precedence sequence is

monadic operators
multiplying operators
adding operators
relational operators.

The programmer is encouraged, however, to make good use of parentheses to avoid error and enhance program readability.

Operators. There are two *monadic* operators. After they have operated on something, the result is an integer, and this result can then be used as any other integer can.

BYTE, followed by a text, yields the integer value of the first character of that text. Example,

BYTE TXT

where TXT was previously defined in the program.

WORD, followed by a text, yields the integer value of the first and second characters of that text, where these two characters are taken together. Thus, if TXT is

1001000111110011

then

BYTE TXT

yields the integer value of 10010001, and

WORD TXT

yields the integer value of 1001000111110011.

The *multiplying* operators are multiplication and division.

The *adding* operators are plus, minus, and the three logical operators SHIFT, EXTRACT, and AND.

SHIFT

A SHIFT 2

shifts A two places to the left, filling the empty righthand positions of A with zeros.

A SHIFT –2

shifts A two places to the right, filling the empty left-hand positions with zeros. SHIFT is not a wrap-around operation. Bits shifted out of the word are lost.

EXTRACT allows the programmer to take a part of the current contents of an integer variable and make that part into an integer.

VAR2: = VAR1 EXTRACT 8;

takes the last eight bits of the variable VAR1 contents and places them in VAR2. VAR1 EXTRACT 8 can also be used by itself as an integer.

AND is the logical 'and'.

VAR1 AND VAR2

yields the integer value of the logical 'and' operation, as performed on the current contents of the previously defined integer variables VAR1 and VAR2.

The programmer should note that division by zero, or the division of zero by zero, will NOT give an error message.

When text strings are compared, the comparison takes place only on the number of characters that is minimum for the pair of strings. That is, in the comparison

IF ALPHA > BETA THEN........

where ALPHA is occupied by

TR

and BETA is occupied by

TRANS

the comparison will consider only the first two characters of BETA, so that in this example ALPHA and BETA are equal.

The programmer should note that the following is NOT allowed:

IF 'THIS IS THE END' = 'THIS IS IT' THEN....

Comparisons can compare on variable *names* only.

I/0 Handling

I/0 operations are performed on files and on parts of those files. In order to identify the file being operated upon, as well as the part of the file that is currently being used, a place is reserved for file descriptors. This description is called the 'zone descriptor'. In the zone descriptor we find

Document name	The name of the driver process, e.g., MTO.
Kind	Information on the type of device. See the Kind Table.
Operation	Defined in the OPEN file instruction. See Operation Mode Table.
GIVEUP mask and address	This is defined in the file declaration.
Blockcount and File count	The current block and file number.
Used Share and Sharelength	Tells what the current share is and the length of the buffer.
Record Format and Length	
First Byte, Top Byte, Remaining Bytes	Contains pointers to the first byte of the current record, the first byte after the current record, the rest of the bytes of the share.
Conversion Table	The conversion table address.

In addition to the Zone Descriptor, the Zone contains Share Descriptors, and a Buffer Area. The Share descriptors contain information about the current activities in the buffers which they describe, and the Buffer Area contains the descriptors and the associated buffers. Certain symbols are provided for operating on the Zone Descriptor. By chosing integers to set into these areas, one can assume total control over I/0 operations. The way this is done is described in the MUS manual, which the programmer should read before attempting to use these expressions, which are to be considered as items available only in advanced MUSIL programming.

The contents of the Zone Descriptor which can be reset by the programmer are

filename . ZMODE	gives the mode of operation, see Operation Mode Table.
filename . ZMASK	is the giveup mask for device errors
filename . ZFILE	is used differently for different devices, see MUS manual
filename . ZBLOCK	may be the current block or the number of blocks done, see MUS manual
filename . ZFIRST	is the byte address of the first byte of the current record
filename . ZTOP	points to the first byte after the current record
filename . ZLENGTH	is the length in bytes of the current record
filename . ZREM	is the length in bytes of the remaining part of the current block
filename . Z0	can occur only inside a GIVEUP procedure, where it tells which errors got you into the procedure

The beginning MUSIL programmer will use only these last three.

In sum, then, the documents that we input to, or output from, our job are described inside the zone descriptor by the document name (which is the process name of the driver that controls the device the document will be on), the operation code that is sent to the driver (telling whether we are operating with input or output, etc., as defined in the Operation Code Table), and device kind (which tells if the device is character or block oriented, if position or repetition are possible, see Kind Table).

Handling Exceptions

In the I/0 procedures, the programmer can choose to determine what should be done at End of Tape, End of File, when parity errors occur, etc. Or the programmer can let the system handle exceptions in its standard way. If it is not desired to let the system do this, then the programmer must write a GIVEUP procedure. In the absence of a GIVEUP procedure, the STATUS word that determines exception handling will be set up in the following way automatically by the system:

bit	event	action
0	deviced disconnected	hard error
1	device off-line	hard error
2	device busy	operation is repeated
3	device mode 1	ignored, defined in Operation Mode Table
4	device mode 2	ignored, defined in Operation Mode Table
5	device mode 3	ignored, defined in Operation Mode Table
6	illegal instr.	hard error
7	EOF	hard error
8	block length error	hard error
9	data late	if kind bit 12 is 1, then the operation is repeated, otherwise, hard error
10	parity error	same as for bit 9
11	end medium	error is hard, except for certain conditions that the MUSIL beginner should not take into account
12	position error	hard error
13	rejected	hard error
14	timer	hard error
15	repeat error	hard error

When a hard errors occurs, processing stops and the error number and unit name are displayed on the operator's console. If the operation is repeated when an error occurs, then there will be a maximum of five repititions, after which time, the error becomes a Repeat Error, and is hard.

In error handling, tne system will perform the treatments described for bits 0 through 11, plus bit 14 first. Then it will look to see if there are 1 bits in the GIVEUP procedure. If there are, then control will be given to the GIVEUP procedure. If not, then a hard error will occur.

The GIVEUP procedures are arranged in a hierarchy of instructions, as follows: For example, the programmer may use an instruction to make space for a record in an output buffer. When the programmer issues this command, which happens to be PUTREC, described below, then the following hierarchy of commands (also described below) becomes involved automatically

```
PUTREC
  ↓
OUTBLOCK ──────────┐
  ↓                ↓
TRANSFER      WAITTRANSFER ←─┐
                  ↓          │
              GIVEUP procedure, if
              specified
```

Here, it should be noted that in MUSIL certain of the I/0 instructions can be redefined by the programmer. The instructions TRANSFER and WAITTRANSFER are used in this way. For the beginning MUSILprogrammer, in the example above the operations specified by TRANSFER and WAITTRANSFER can be left to the system to perform automatically.

If the programmer wishes to have the operator informed of what is in the STATUS word, or in part of it, then the use of the OPSTATUS command is recommended.

Assuming that there is in the Constant Section an definition of what is to be displayed, the instruction is

OPSTATUS(IN.ZO,ERRORS);

where we have previously defined, for example

```
ERRORS = 'DISCONNECT<10><0>
          OFFLINE<10><0>
          .
          .
          .
          TIMER <10><0>
          BIT 15 ?? <10>'
```

IN.ZO is the system-defined expression that contains the STATUS word for the file called IN. For this example, if IN.ZO contains 1000000000000 000, then DISCONNECT will be printed on the operator's console, along with skipping to a new line. If IN.ZO contains 1000000000000010, then

DISCONNECT
TIMER

will be output to the console, along with skipping to a new line, etc.

I/0 Instructions

OPEN (filename, mode)

The file name should have been defined in the VARIABLE Section, and the mode can be defined by reference to the Operation Mode Table, for it will be different for different devices.

This instruction opens the file and sets various pointers. If in the body of the program we wish to identify or change the mode, then we can access it by

filename . ZMODE

CLOSE (filename, release)

If release is not equal to zero, then the device will be released to another program. If, for example, we are working with magnetic tapes then the tape will first be rewound and set off-line.

If we do not want the tape to be rewound, then we set release to 0 . This results in a file mark being written. The exact sequence of events for other devices can be found in the MUS manual.

WAITZONE (filename)

This command allows one to interrupt I/0 processing in an orderly way. The information needed for continuing with the processing later on is stored, so that one can resume processing wherever one wants. Suppose we have

```
IF operator action THEN
   BEGIN
     WAITZONE (filename);
       interrogate operator
   END;
```

The WAITZONE lets the communication take place in such a way that processing can be resumed in an orderly way after the communication.

SETPOSITION (filename, file number, block number)

This instruction automatically calls WAITZONE. Then it positions the I/0 medium, such as MT0 for example, and finds the number of the file and block within it that processing will start on. For example,

SETPOSITION (MT0, 3, 8)

positions processing to the 8th block of the 3rd file within MT0.

GETREC (filename, variable name)

Example: GETREC(INFILE, SIZE);

The events that this instruction cause depend on the record format:

For undefined (in file definition) format and unblocked. This instruction gets the next physical block. It is much used for reading cards, for in this case it reads the next card. When used, say, with paper tape, it would read as much of the tape as there is room for in the buffer. At call time SIZE is irrelevant. At execution time the system will put the size of the block read into SIZE.

For undefined and unblocked.
The number of characters equal to SIZE' will be read. This means that you can read, say, the first byte of a magnetic tape block. This can be done thus:

```
SIZE = 1;  –
GETREC(MT0, SIZE);
```

If during read-in the GETREC command is used with SIZE greater than the remaining part of the block, then the system will begin to read the next block. If we write

```
SIZE: = 1;
GETREC(MT0,SIZE);
IF BYTE MT0 = binary code THEN
      BEGIN
        SIZE: = MT0.ZREM;
        GETREC(MT0,SIZE);
             .
             .
             .
             .
        processing of block

        END;
```

Then what we have done is, first, inspect-the first byte of the tape block to see what sort of block it is, then, read in the remainder of the block (ZREM) and processed it.

For fixed length and unblocked.
In this case the record has previously been defined. GETREC causes the next physical block to be read, taking as many bytes as were specified in the record definition and skipping the remaining bytes in the block. The system will put into SIZE the number of bytes read in.

For fixed length and blocked.
The system looks to see if the current block contains the next record. If so, it reads it. If not, it goes to the next block. (Throughout, it should be kept in mind that 'unblocked' means that the block is not divided into records.)

For variable length and unblocked.
The next block is read. The first four bytes, containing the block length, are decoded. The next four bytes, containing the record length, are decoded. The record length is put into SIZE: For all practical purposes, we are here talking about IBM V format magnetic tapes.

For variable and blocked. IBM VB format.
The next record from the current block is read by decoding the first four bytes. If there is no record left in the current block, the first record of the next block is read.

PUTREC (filename, name or number or expression)

The events cause by this command depend on record structure.

For undefined and unblocked.
The previous block is output. If we say PUTREC(FILENAME,SIZE), then space is reserved in core for SIZE bytes of the next block to be output the next time PUTREC is called.

For undefined and blocked.
The system looks to see if the current physical block in core can contain yet another record of SIZE bytes. If so, it makes room for that additional record. If not, it outputs the current block.

For fixed and unblocked.
The current block is output and space is reserved for the next record. SIZE is irrelevant, as it was given in the record definition.

For fixed and blocked.
Events are as in unformatted and blocked, except that SIZE is irrelevant, having been given in the record definition.

For variable and unblocked.
reserved for the next record. The four-byte block size and the fourbyte record size are computed and put into the block. This allows such output to be read later on by a GETREC in V format.

For variable and blocked.
The system checks to see if there is room for the next record, as determined by SIZE. If so, it makes a four-byte record descriptor word and puts it on the record. Then data can be put in. Finally, the block descriptor word is up-dated. If not, the block is output.

If the file is undefined and unblocked, then the following two instructions can be used.

INCHAR(filename, integer variable name)
puts the next byte from the file into the integer variable name. If there are no bytes left in the current block, the first byte to the next block is used.

OUTCHAR(filename, constant)
checks to see if there is room for a byte in the current block. If so, it puts a byte into the block. If not, it puts the byte into the next block. The byte that is put into the block is whatever is in the first byte of the constant. The constant may be a number, the current contents of a variable, or the contents of an expression:

```
OUTCHAR(OFILE,54);
OUTCHAR(UFILE,VAL);
OUTCHAR(FL,X+Y);
```

OUTTEXT(filename, string variable name)
outputs the string contained in the string variable until a final binary zero is reached,which means that the string must contain such a binary zero.

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

Example
```
MOVE(IN↑, 1, OUT↑,0 , LENGTH);
```
This example takes the current input record, starting with the second byte, and moves it into the current output record, starting with the first byte. The number of bytes moved is equal to the number in LENGTH. Note that if LENGTH is too big, there will be no error message. Finally, MOVE *cannot* be used to move bytes around within the same string.

CONVERT(string name, string name, table name, length)
This instruction is used to convert between media, such as between magnetic tape and teletype, 7- and 9-track magnetic tape, etc.

Example:
```
CONVERT(MT0↑,OUT↑,TABLE1,OUT:ZLENGTH);
```

This example takes the current record of MT0 and converts it according to TABLE1, and puts the result into the current record of OUT. It does this for as many bytes of the record as is the numerical value of length, which in this case is the length of the current OUT record. Length could be an expression, a number, or a variable.

TRANSLATE (byte name, byte name, table name)
This instruction converts the first byte, using the table, and puts the result into the second byte.

Example:
```
TRANSLATE(IN↑,CCW,OUT↑,CCW,ANSITABLE);
```
which converts a byte of file IN and places the result in the appropriate byte of file OUT. If the system cannot find an argument in the table, then it will put out the default value. The table should have been organized thus:

```
CONST
  .
  .
ANSITABLE = #arg1   value1
             arg2   value2
              .
              .
              .
             0        0
             0      default value#
```

Note the three zeros which preceed the default value. They are required. Note also that it is good programming practice to put each argument/value pair on a separate line for easy reading.

INSERT (byte name, record name, place)

Example:
```
INSERT(SP SHIFT 5-1, OUT  , OUT.ZLENGTH -1);
```
This instruction takes the 8 least significant bits of the first-named byte and puts them into the place specified in the second-named record.To put the byte into the first place of the record, write something like

```
INSERT(A,B,0);
```

REPEATSHARE (filename)
This instruction is used *only* within a GIVEUP procedure. In case of error, it will repeat the operation that gave rise to the error message. Obviously, its use can accidentally give rise to an unending operation, if the programmer is not careful. The following example illustrates its use.

```
PROCEDURE GENERALGIVEUP
     BEGIN
       OPMESS(SOMETHING WRONG);
       message to operator console
       OPIN(OPSTRING);
       operator perform action
       OPWAIT(OPSTRING);
       wait for operator action
      REPEATSHARE (filename)
      END;
```

We have now completed the description of the MUSIL commands that the beginning MUSIL programmer should be familar with.

In addition to the commands described so far, there are additional commands that can be used by the experiencedMUSILprogrammer. A complete description of the effects of these commands can be found in the MUSIL reference manual. Before attempting to use these commands, however, the programmer should familiarize himself with the MUS operating system and its instruction set. For completeness' sake, we shall now mention four of the most common advancedMUSILcommands.

INBLOCK (filename)
This instruction is used for coding one's own GETREC or INCHAR. It is not meant for the beginner. The instruction GETs a block.

OUTBLOCK (filename)
This instruction is used to code one's own PUTREC or OUTCHAR. It is not meant for the beginner .It readies a buffer for output.

TRANSFER (filename, length, operation)
This instruction should not be used by the beginner. It is used for coding one's own INBLOCK and OUTBLOCK operations. "Length" is the maximum number of bytes to be input or output. "Operation" is a 16-bit code (found in the MUS manual) telling the driver what to do.

WAITTRANSFER (filename)
This instruction is used with the above. It should not be used by beginning MUSILprogrammers.

SPECIAL WORDS
The following words have special meanings in MUSILThey should not be used by the programmer for naming variables, constants, tables, or procedures, even though in many cases no harm would be done.

AND	LENGTH	TRANSLATE
BEGIN	MOVE	TYPE
BINDEC	MUSIL	U
BYTE	OF	UB
CLOSE	OPEN	UNTIL
CONST	OPIN	V
CONV	OPMESS	VAR
CONVERT	OPSTATUS	VB
DECBIN	OPTEST	WAITTRANSFER
END	OPWAIT	WAITZONE
EXTRACT	OUTBLOCK	WHILE
F	OUTCHAR	WORD
FB	OUTTEXT	ZBLOCK
FILE	PROCEDURE	ZFILE
FROM	PUTREC	ZFIRST
GETREC	RECORD	ZLENGTH
GIVEUP	REPEAT	ZMASK
GOTO	REPEATSHARE	ZMODE
IF	SETPOSITION	ZREM
INBLOCK	SHIFT	ZTOP
INCHAR	STRING	ZO
INSERT	THEN	
INTEGER	TRANSFER	

RELEASE TABLE
- 0 driver is not released for another program
- 1 driver is released

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

Examples:

1110	Magnetic tape station
0001	Line printer
0011	Line printer
0010	Card reader
0001	Teletype
0001	Paper tape punch
0001	Paper tape reader

Operation Code

The operation code is the 2 least significant bits of the operation mode.

Operation Mode Table

Paper tape reader driver
- 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
- 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
- 3 the converted characters are output
- 7 the first byte of output is interpreted as a carriage control word

Magnetic tape driver
- 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

Card reader driver
- 5 read binary punched cards
- 21 read decimal punched cards
- 33 read decimal punched cards and skip trailing blank columns (a minimum of ten columns are read)

The operation mode designators for the other available RC 3600 I/0 devices can be found in the MUSIL reference manual. The above devices are the only ones that the beginning MUSIL programmer should concern himself with.

Error Messages

MUSIL provides the programmer with a variety of error messages, indicated by error numbers on the compilation printout. The significance of those error numbers is as follows:

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 ident.
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.

Musil Program Example

The following program should help you to see how the various MUSIL instructions can be put together to form a complete program.

```
!

                                      PROGRAM RC36-00001.00

                                      MUS PRINT IMAGE

KEYWORDS:                 MUSIL,CONVERSION,MTA,LPT,LISTING

ABSTRACT:                 THIS PROGRAM HANDLES NO LABEL TAPES WITH A
                          MAXIMUM BLOCK SIZE OF 1340 BYTES, EACH BLOCK
                          CONSISTING OF FIXED LENGTH RECORDS WITH CCW
                          CONTROL CHARACTERS AND EBCDIC CODE DATA.
                          OUTPUT ON RC3600 SERIES PRINTERS WITH 64
                          CHARACTER ASCII DRUM.
                          THE PROGRAM MAY BE OPERATED FROM EITHER OCP OR TTY.

RCSL 43-GL103:            ASCII SOURCE TAPE         !
```

```
!

TITLE:                 MUS PRINT IMAGE.

ABSTRACT:              THIS PROGRAM HANDLES NO LABEL TAPES WITH A MAXIMUM
                       BLOCK SIZE OF 1340 BYTES, EACH BLOCK CONSISTING OF
                       FIXED LENGTH RECORDS WITH CCW CONTROL CHARACTERS
                       AND EBCDIC CODE DATA. OUTPUT ON RC3600 SERIES PRIN-
                       TERS WITH 64 CHARACTER ASCII DRUM.
                       THE PROGRAM MAY BE OPERATED FROM EITHER OCP OR TTY.

SIZE:                  5674 BYTES.
DATE:                  JULY 29TH 1974.

RUNTIME PARAMETERS:
     BLOCK NO : 00001     NEXT BLOCK TO BE READ FROM CURRENT FILE.
     FILE NO  : 00001     THE FILE FROM WHICH THE BLOCK IS READ.
     REWIND   :   +       INDICATES IF REWIND OF TAPE AT EOF.
     MARGIN   : 00000     SPACES TO THE LEFT OF THE PRINT LINE.
     SELECT   : 00999     DEFAULT CCW SWITCH, SELECT MODE/VALUE.
     RECSIZE  : 00133     LENGTH OF INPUT RECORD.

OTHER OUTPUT MESSAGES:
     CONTSTATE:  +/-      STATE OF CONTINUE SWITCH (TTY ONLY).
     PROG NO  :      1    PROGRAM EXECUTION IS STOPPED.
     RUNNING              PROGRAM EXECUTION IS STARTED.
     SUSPENDED            DRIVERS RELEASED, PROGRAM EXECUTION IS STOPPED.
     MOUNT DATA TAPE      MT-UNIT IS NOT ON-LINE.
     MT ERROR   00022     MT-UNIT IS REWINDING.
     MT ERROR   00023     NOISE RECORD.
     MT ERROR   00026     MT DRIVER RESERVED.
     MT ERROR   00028     BLOCK LENGTH ERROR.
     MT ERROR   00029     DATA LATE.
     MT ERROR   00030     PARITY ERROR.
     MT ERROR   00034     TIME OUT AT WAITINTERRUPT.
     LP ERROR   00021     LP IS OFF-LINE.
     LP ERROR   00026     LP DRIVER RESERVED.
     LP ERROR   00028     BLOCK ERROR, PAPER FAULT, PAPER RUN AWAY.
     LP ERROR   00029     DATA LATE.
     LP ERROR   00030     CCW PARITY ERROR.
     LP ERROR   00031     PAPER LOW.
     LP ERROR   00034     TIME OUT AT WAITINTERRUPT.
     END JOB              PROGRAM EXECUTION IS TERMINATED.

INPUT MESSAGES:
     STOP                 STOPS EXECUTION WRITING PROG NO  :      1.
     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 MT ERROR START MEANS ACCEPTING
                          THE ERRONEOUS INPUT, AFTER LP ERROR START
                          MEANS REPEATING THE PRINT OPERATION.

SPECIAL REQUIREMENTS:     NONE.
!
```

```
CONST
NOQ=                7,
OPTXTS=
'<14><6>
<10>PROG NO   :      1<0>
<10>BLOCK NO : <0>
<10>FILE NO   : <0>
<10>REWIND    :   <0>
<10>SELECT    : <0>
<10>MARGIN    : <0>
<10>RECSIZE   : <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>',
MTTXT=              '<7><10>MT ERROR    ',
LPTXT=              '<7><10>LP ERROR    ',
EOJTXT=             '<14><7><10>END JOB<13><0>',
SUSTXT=             '<7><10>SUSPENDED<13><0>',
MTMOUNTTAPE=        '<14><7><10>MOUNT DATA TAPE<13><0>',
CONTSTATE=          '<10>CONTSTATE:    <0>',
```

```

LPTABLE=          ! EBCDIC TO 64 CHARACTER ASCII DRUM
                  0   1   2   3   4   5   6   7 !
#
! 0 !   255 255 255 255 255 255 255 255
! 8 !   255 255 255 255 255 255 255 255
! 16 !  255 255 255 255 255 255 255 255
! 24 !  255 255 255 255 255 255 255 255
! 32 !  255 255 255 255 255 255 255 255
! 40 !  255 255 255 255 255 255 255 255
! 48 !  255 255 255 255 255 255 255 255
! 56 !  255 255 255 255 255 255 255 255
! 64 !  255 255 255 255 255 255 255 255
! 72 !  255 255  91  46  60  40  43  92
! 80 !   38 255 255 255 255 255 255 255
! 88 !  255 255  33  36  42  41  59  93
! 96 !   45  47 255 255 255 255 255 255
! 104 ! 255 255 255  44  37  94  62  63
! 112 ! 255 255 255 255 255 255 255 255
! 120 ! 255 255  58  35  64  39  61  34
! 128 ! 255  65  66  67  68  69  70  71
! 136 !  72  73 255 255 255 255 255 255
! 144 ! 255  74  75  76  77  78  79  80
! 152 !  81  82 255 255 255 255 255 255
! 160 ! 255 255  83  84  85  86  87  88
! 168 !  89  90 255 255 255 255 255 255
! 176 ! 255 255 255 255 255 255 255 255
! 184 ! 255 255 255 255 255 255 255 255
! 192 ! 255  65  66  67  68  69  70  71
! 200 !  72  73 255 255 255 255 255 255
! 208 ! 255  74  75  76  77  78  79  80
! 216 !  81  82 255 255 255 255 255 255
! 224 ! 255 255  83  84  85  86  87  88
! 232 !  89  90 255 255 255 255 255 255
! 240 !  48  49  50  51  52  53  54  55
! 248 !  56  57 255 255 255 255 255 255
#;

```

```
VAR
OPDUMMY:         STRING(2);      ! RUNTIME PARAMETERS !
PROGNO:          INTEGER;
BLOCKNO:         INTEGER;
FILENO:          INTEGER;
REWIND:          INTEGER;
SELECT:          INTEGER;
MARGIN:          INTEGER;
RECSIZE:         INTEGER;
OPTEXT:          STRING(20);     ! COMMUNICATION AREA !
OPSTRING:        STRING(20);
OPDEC:           STRING(10);
OPCONT:          STRING(2);      ! INTERNAL VARIABLES !
NEXTCONT:        STRING(1);
GLCONT:          STRING(1);
CURZZZ:          STRING(1);
SELX:            INTEGER;
SELY:            INTEGER;
SELZZZ:          INTEGER;
DATAINDEX:       INTEGER;
SELECTINDEX:     INTEGER;
ERRORNO:         INTEGER;
MASK:            INTEGER;
TOM:             INTEGER;
SIGN:            INTEGER;
Q:               INTEGER;
PAR:             INTEGER;
LENGTH:          INTEGER;
RECLENGTH:       INTEGER;
LPLENGTH:        INTEGER;
LPDATALENGTH:    INTEGER;
P1:              INTEGER;
P2:              INTEGER;
P3:              INTEGER;
S1:              STRING(2);
S2:              STRING(2);
NEXTLP:          INTEGER;
```

```

IN:     FILE                            ! INPUT FILE DESCRIPTION !
        'MTO',                          ! NAME OF INPUT DRIVER !
        14,                             ! KIND= REPEATABLE, !
                                        !       POSITIONABLE, !
                                        !       BLOCKED. !
        1,                              ! BUFFERS !
        1340,                           ! SHARESIZE !
        FB;                             ! FIXED BLOCKED !

        GIVEUP
        MTINERROR,                      ! MT ERROR PROCEDURE !
        2'0110001111011011              ! GIVE UP MASK !
                                        ! ALL REPEATABLE BITS OFF !
                                        ! AND BIT 15 ON !

        OF RECORD                       ! RECORD STRUCTURE !
           CCW:        STRING(1);
           SELECT1:    STRING(1)        FROM 1;
           DATA:       STRING(1)        FROM 1;
           SELECT2:    STRING(1)        FROM 2
        END;

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

        GIVEUP
        LPERROR,                        ! LP ERROR PROCEDURE !
        2'1100001011110010;             ! GIVE UP MASK !

        CONV
        LPTABLE                         ! CONVERSION TABLE !

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

```

```
PROCEDURE INITPOSITION;
BEGIN
    IF IN.ZMODE=0 THEN OPEN(IN,1);
    IF OUT.ZMODE=0 THEN OPEN(OUT,7);
    SETPOSITION(IN,FILENO,BLOCKNO);
    SETPOSITION(OUT,MARGIN,0);
    IN.ZLENGTH:=RECSIZE;
    SELX:=SELECT/10000;
    SELY:=(SELECT-SELX*10000)/1000;
    SELZZZ:=(SELECT-SELX*10000)-SELY*1000;
    DATAINDEX:=1-SELX;
    IF SELZZZ<256 THEN DATAINDEX:=DATAINDEX+1;
    SELECTINDEX:=DATAINDEX-1;
    LPLENGTH:=RECSIZE-DATAINDEX+1;
    IF LPLENGTH+MARGIN>133 THEN LPLENGTH:=133-MARGIN;
    LPDATALENGTH:=LPLENGTH-1;
END;

PROCEDURE CONTINUE;
BEGIN
    GLCONT:=OPCONT;
    OPCONT:=NEXTCONT;
    NEXTCONT:=GLCONT;
    UPMESS(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 LENGTH > 6 THEN DIRECTUPDATE;
                    IF LENGTH > 6 THEN GOTO 1020;
                    IF OPTEXT = START THEN GOTO 1070;
                    IF OPTEXT = STOP  THEN GOTO 1000;
                    IF OPTEXT = SUSPEND THEN GOTO 9;
                    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 GOTO 9;
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 MTINERROR;
BEGIN
    IF IN.Z0 AND 256 <> 0 THEN !EOF! GOTO 9;
    IF IN.Z0 <> 8'001000 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 OPTEXT=START;
    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);
        BLOCKNO:=IN.ZBLOCK;
        MASK:=OUT.Z0;
        SHOWERROR;
        NEXTLP:=0;
        REPEAT OPSTOP UNTIL OPTEXT=START;
        OPMESS(RUNTXT);
        IF OUT.Z0 AND 8'141342 <> 0 THEN
        REPEATSHARE(OUT);
    END;
END;
```

```
     BEGIN
     IN.ZBLOCK:=1; BLOCKNO:=1; FILENO:=1; REWIND:=-1;
     SELECT:=999; MARGIN:=0; RECSIZE:=133; NEXTLP:=0;
     OPCONT:=FIFTEEN; NEXTCONT:=FIVE; OPIN(OPSTRING);
1:   OPCOM;
     INITPOSITION; IF OPTEST<>0 THEN OPSTOP;

2:   REPEAT BEGIN
          GETREC(IN,RECLENGTH);
          IF SELZZZ<256 THEN
          BEGIN
               MOVE(IN↑.DATA,SELECTINDEX,CURZZZ,0,1);
               IF SELY=0 THEN
               BEGIN
                    IF BYTE CURZZZ<>SELZZZ THEN GOTO 5;
                    GOTO 3;
               END;
               IF BYTE CURZZZ AND SELZZZ=0 THEN GOTO 5;
          END;
3:        PUTREC(OUT,LPLENGTH);
          IF SELX=0 THEN
          BEGIN
               OUT↑.CCW:=IN↑.CCW;
               GOTO 4;
          END;
          OUT↑.CCW:=SP1A;
4:        MOVE(IN↑.DATA,DATAINDEX,OUT↑.DATA,0,LPDATALENGTH);
5:   END UNTIL IN.ZREM<RECSIZE;
     BLOCKNO:=IN.ZBLOCK;
     IF OPTEST=0 THEN GOTO 2;
     WAITZONE(OUT);
     OPSTOP; GOTO 2;

9:   CLOSE(OUT,1);
     IF OPTEXT=SUSPEND THEN
     BEGIN
          CLOSE(IN,1);
          OPMESS(SUSTXT);
          GOTO 10;
     END;
     BLOCKNO:=1; FILENO:=FILENO+1;
     IF OPCONT = FIVE THEN
     BEGIN
          CLOSE(IN,1);
          FILENO:=1;
          OPMESS(MTMOUNTTAPE);
     END;
     IF OPCONT = FIFTEEN THEN
     BEGIN
          CLOSE(IN,REWIND+2);
          IF REWIND=-1 THEN FILENO:=1;
          OPMESS(EOJTXT);
     END;
10:  REPEAT OPSTOP UNTIL OPTEXT=START;
     INITPOSITION; OPMESS(RUNTXT); GOTO 2;

     END;
SIZE: 02837
```