AAGAARD
RC MANUELS

Title:

MUSIL Codeprocedure Library
Programmer's Reference.

A/S REGNECENTRALEN

RC SYSTEM LIBRARY: FALKONERALLE 1 DK-2000 COPENHAGEN F

RCSL No: 43-GL7059
Edition: May 1978
Author: Klaus Hansen

Keywords:

RC3600, MUS, MUSIL, Codeprocedure Library, Programmer's Reference

Abstract:

This manual is a reference for MUSIL programmers using existing codeprocedures.

Printed by A/S Regnecentralen, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

CONTENTS

CONTENTS (continued) PAGE

1. PREFACE.

This manual is intended to supply information about codeprocedures in the library of common interest for MUSIL programmers.

The set of procedures described is not exhaustive, as some procedures are written to specific applications, and consequently perform functions too special for general use. Programmers are referred to the listings of these procedures.

The division of the procedure library into four major groups (corresponding to chapter 2-5) is somewhat arbitrary as some procedures may belong naturally to more than one group, e.g. SPRIO (3.3.3.) and STOPPROCESS (2.3.2.).

Chapter 6 is an exhaustive list of all codeprocedures currently in the library.

2. MONITOR FUNCTIONS.

This chapter describes procedures making various MUS monitor functions and DOMUS functions available for MUSIL programs.

2.1. Send Message, Wait Answer.

See 2.2.1., 2.2.2. in addition to 2.1.1.

2.1.1. Send Message.

Declarations:

```
procedure SMESSAGE        (const   INDEX:    integer;
                           const   DNAMES:   string(1);
                           var     MESS0:    integer;
                           var     BUF:      integer     );

codebody;
procedure SENDMESSAGE     (var     BUF:      integer;
                           const   NAME:     string(6);
codebody P0164;            const   MESS:     string(8)   );
```

The procedure SMESSAGE sends the message contained in MESS0 and following 3 integers to the process with name in DNAMES (INDEX) and following 5 bytes; unused bytes in the name shall be zero. The message buffer address is returned in BUF, for later use in AANSWER or TESTMESSAGE.

The 4 integers containing the message shall be declared together, e.g.

```
MESS0,MESS1,MESS2,MESS3 : integer;
```

The procedure has no P.. number.

The procedure SENDMESSAGE performs a similar function, the name given in NAME and the four word message in MESS, which may be declared like

```
MESS:   record
           MESS0,
           MESS1,
           MESS2,
           MESS3   :  integer

        end;
```

2.1.2. Wait Answer.

2.1.2.

Declaration:

```
procedure AANSWER        (const   BUF:     integer;
                          var     ANSW0:   integer     );

codebody;
```

The procedure accepts an answer to the message given by BUF, by means of the MUS call WAIT ANSWER. The answer is delivered in ANSW0 and the following 3 integers, as for SMESSAGE.

The procedure has no P.. number.

2.1.3. Test Message State.

2.1.3.

Declaration:

```
procedure TESTMESSAGE    (const   BUF:     integer;
                          var     RESULT:  integer     );

codebody;
```

The procedure stores BUF.receiver in RESULT:

RESULT = 0	free
> 0	message not yet answered
< 0	answer to the message ready for use by AANSWER.

No checking is done whether BUF points to a message buffer. BUF may be set by SMESSAGE.

The procedure has no P.. number.

2.2. Wait Message, Send Answer.

2.2.1. Getevent.

Declaration:

```
procedure GETEVENT       (var     SENDER:     integer;
                          var     BUF:        integer;
                          const   WAITTIME:   integer;
                          var     EVENT:      integer     );
```

codebody P0047;

The procedure waits for any event, with timer.

Call parameters:

SENDER	0	Any event accepted
	<>0	Only events with event.sender equal to SENDER are considered.
BUF	0	The event queue is examined from its beginning.
	<>0	The event queue is examined starting after BUF.

WAITTIME	0	Return latest 20 ms after inspection of the event queue.
	-1	Wait until event arrives.
	1-13107	Wait 100 ms - 21 sec if no event arrived.

Return parameters:

SENDER	0	No event, i.e. timer runout.
	<>0	Event arrived with event.sender, which is returned in SENDER.
BUF		If no event arrived, BUF in unchanged, otherwise it points to the event.
WAITTIME		Unchanged.
EVENT		If no event arrived, EVENT is unchanged, otherwise EVENT and the following 3 integers contain the four words of the event.

2.2.2. Waitmessage. 2.2.2.

Declaration:

```
procedure WAITMESSAGE  (var     MESS0:    integer;
                         var     SENDER:   integer;
                         var     BUF:      integer     );
```

codebody P0156;

Waits for an event by means of the MUS call WAITEVENT.

Call values:

BUF:	0	The event queue is inspected from the beginning.
	<>0	The event queue is inspected starting with the element after BUF.

Return values:

BUF — Contains the event address.

SENDER — Set to event.sender.

MESS0 — Contains the first word of the event. The three remaining are placed in the following 3 integers.

2.2.3. Send answer.

Declaration:

```
procedure SENDANSWER     (var     ANSW0:   integer;
                          var     BUF:     integer     );
codebody P0040;
```

The procedure sends an answer to the message specified by BUF, using the MUS call SENDANSWER.

The answer contents is taken from ANSW0 and the following 3 integers.

No checking is done of the validity of BUF. A correct value of BUF may be obtained by means of WAITMESSAGE (2.2.2.).

2.3. Miscellaneous.

2.3.1. Delay.

Declaration:

```
procedure DELAY          (const   INTERVAL:   integer);
codebody P0023;
```

The procedure causes a delay of INTERVAL x 0.1 sec (by means of waitinterrupt). INTERVAL should be 1-13107.

2.3.2. Stop/Start of a Process. 2.3.2.

Declaration:

```
procedure STOPPROCESS;
codebody P0048;

procedure STARTPROCESS  (const   NAME:   string(6)  );
codebody P0165;
```

The procedure STOPPROCESS executes a MUS STOP PROCESS with the effect of stopping execution of the calling program. A START command to MUS will cause execution to be resumed after the procedure call.

The procedure STARTPROCESS starts the process with name given in NAME, unused bytes set to zero. If the process is not found, no action is taken.

2.3.3. Search MUS item. 2.3.3.

Declaration:

```
procedure SEARCHITEM    (const   NAME:    string(6);
                          var    RESULT:  integer     );
Codebody P0007;
```

The procedure searches for a process with the name given in NAME. Unused bytes in NAME should contain zero. The result of the search is returned in RESULT:

```
RESULT  =  0  not found
RESULT  = -1  found
```

2.3.4. Get DOMUS Core Item.

Declaration:

```
procedure GETCOREITEM    (const   NAME:    string(6);
                          var     ADDRESS: integer;
                          var     SIZE:    integer    );
codebody P0117;
```

The procedure searches for a DOMUS core item with the name given in NAME; unused bytes in NAME should be zero. If the core item is not found or belongs to another process, a value of zero is returned in SIZE.

If the core item is found and belongs to the calling process, the size and start address (wordaddress) is returned in SIZE and ADDRESS respectively, including the 7 word header.

2.3.5. Find Process Descriptor.

2.3.5.

Declaration:

```
procedure FINDPROCESS    (var     PROCNAME:   string(6);
                          var     PROCADDR:   integer    );
codebody;
```

The procedure searches the MUS process chain for a process with the name given by PROCNAME. The unused bytes should contain zero. The result is in PROCADDR:

0	Not found
<>0	Word address of process descriptor

The procedure has no P.. number.

See further 3.3.3., SPRIO.

3. PROGRAM ENVIRONMENT. 3.

This chapter contains descriptions of procedures used to communicate with and supply information about the program environment at running time.

3.1. Date and Time. 3.1.

These procedures use the TIME process.

3.1.1. Get the Date. 3.1.1.

Declarations:

```
procedure GETDATE        (var     DATE:     string(8)  );
codebody P0150;

procedure GETDATE        (var     DATE:     string(6)  );
codebody P0151;
```

These procedures use the TIME process (43-GL4925). A message is sent to TIME requesting current date. The result is returned in the parameter DATE. Two formats are available

result when:	TIME loaded	TIME not loaded
P0150	YY.MM.DD	eight spaces
P0151	YYDDD<0>	00001<0>

where	YY	last two digits of year
	MM	month
	DD	day in month
	DDD	day in year.

3.1.2. Compute Date.

Declaration:

```
procedure COMPUTEDATE    (const   OLDDATE:    string(6);
                          const   DAYS:       integer;
                          var     NEWDATE:    string(6)  );
codebody P0011;
```

The dates are given in the format YYDDD (as returned by GETDATE, 3.1.1.). The parameter DAYS gives the number of days to be added to OLDDATE to give NEWDATE. DAYS may be negative.

Leap years are treated correctly.

3.1.3. Get the Time.

Declaration:

```
procedure GETTIME        (var    TIME:    string(8)    );
codebody P0149;
```

The procedure uses the TIME process (43-GL4925). A message is sent requesting the time of day. The result is returned in TIME, in the format

TIME loaded	TIME not loaded
HH.MM.SS	8 spaces

where
HH hour of day
MM minutes
SS seconds.

3.1.4. Compute Time Difference. 3.1.4.

Declaration:

```
procedure COMTIME        (const   STARTTIME:   integer;
                          const   TERMTIME:    integer;
                          var     MINUTES:     integer    );
codebody P0181;
```

The procedure computes the number of minutes elapsed between STARTTIME and TERMTIME, which gives the starting time and finishing time, respectively. The two times are doubleword MUS internal clock values, the most significant bits in the parameter, and the least significant bits in the integer following it in the MUSIL declarations.

3.2. DOMUS Procedures. 3.2.

The DOMUS procedures FINIS (P0084), GETPARAMS (P0085) and CONNECTFILE (P0086) are described in DOMUS User's Guide, part two.

3.3. Own Process Descriptor. 3.3.

3.3.1. Get Process Name. 3.3.1.

Declaration:

```
procedure GETCURNAME    (var     NAME:    string(6)  );
codebody P0055;
```

The processname is delivered in NAME.

3.3.2. Get Operator Name.

Declaration:

```
procedure OPERATOR       (var      NAME:     string(6)  );
codebody;
```

The operator name, as generated by the MUSIL compiler, is returned in NAME, starting in an even byte address.

3.3.3. Set Priority.

Declaration;

```
procedure SPRIO          (const    NAME:     string(6);
                          var      ADDR:     integer;
                          const    PRIO:     integer    );
codebody;
```

If NAME is the name of a process, the priority of this process is changed to PRIO (if not zero), and ADDR is set to the process descriptor address. Otherwise ADDR is set to zero.

Currently typical priorities are:

8'200	MUSIL program
8'100000	Driver.

3.4. DISC Catalog.

3.4.

3.4.1. Lookup Disc Unit Description.

3.4.1.

Declaration:

```
procedure LOOKUPCATN     (const   UNIT:    string(6);
                          var     DESCR:   string(32)   );
codebody P0132;
```

The parameter UNIT identifies the disc unit in question, and has the format 'UNIT<unitno><0>', i.e. the fifth byte contains a binary value from 0 to 255.

If the program item 'CATW' is present and contains a description of the unit, this description is returned in DESCR as follows:

Byte no	contents
0 - 5	Catalog name, as given in UNIT.
6 - 11	Disc driver name.
12 - 15	Disc displacement, double word integer.
16 - 17	Normal slice size.
18 - 19	Increment slice size.
20 - 21	Number of segments in catalog.
22 - 23	Number of free segment in catalog.
24 - 25	First data segment.
26 - 27	Top data segment.
28 - 29	Min. slice.
30 - 31	Max. slice.

3.4.2. Set Catalog Entry.

Declaration:

```
procedure SETENTRY        (file    F;
                           const   ENTRY:   string(32)   );
codebody P0154;
```

The procedure sets an entry in a CAT76 catalog by means of the MUS call SETENTRY.

3.5. Own Message Buffer Pool.

Declaration:

```
procedure CREATEMESSBUFS  (var     BUFAREA:   string(1);
                           const   LENGTH:    integer    );
codebody P0054;
```

The procedure creates additional message buffers. It has only effect the first time called from a program.

LENGTH is given in bytes. The part of BUFAREA defined by
    BUFAREA(0) .. BUFAREA (LENGTH-1)
is divided into LENGTH//20 message buffers, which are linked into the chain of free, unused message buffers.

The fields mess0 .. mess3 are initialized with the 8 byte text
    'NOT USED'
which may be convenient in connection with core dumps.

4. FILES.

These procedures extend the set of standard MUSIL filehandling procedures.

4.1. Initialize Shares.

Declaration:

```
procedure INITZONE       (file    Z;
                          const   SHARES:    integer;
                          const   LENGTH:    integer;
                          const   AREA:      integer  );
codebody P0155;
```

The file Z is declared like

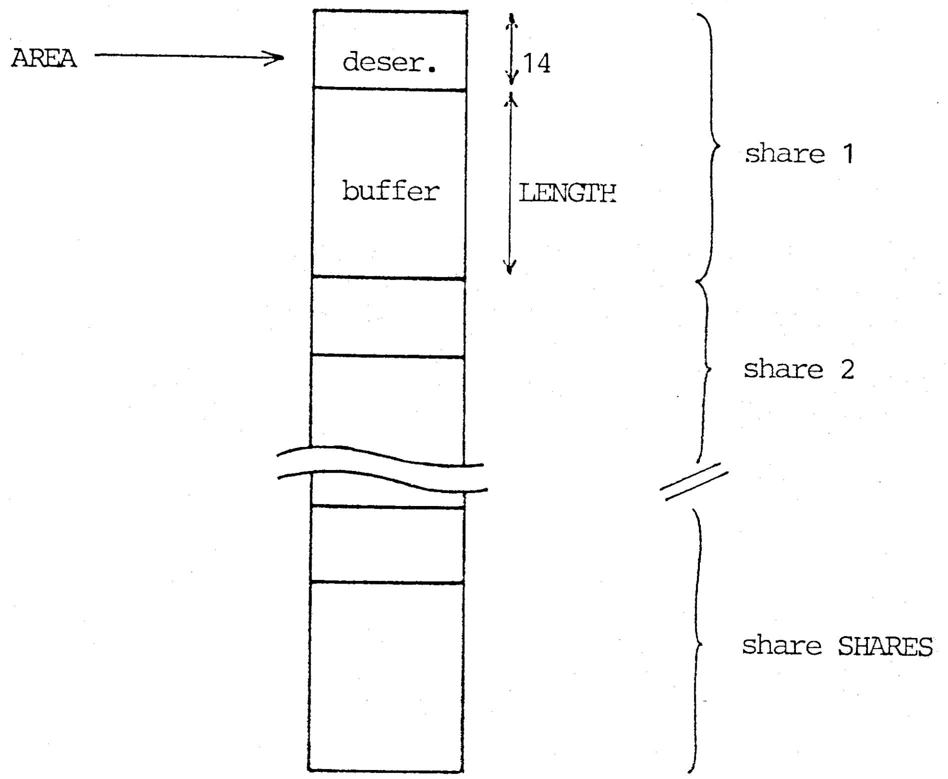
```
     Z:  file ..., ..., 1, 1 ...
         ...
         of ... ;
```

i.e. with 1 share and a buffer of 1 byte.

The value of AREA is an absolute byte storage address, e.g. obtained by a call of TAKEADDRESS (5.6.1.), and shall be even. The parameter SHARES gives the number of shares to be created. The parameter LENGTH is the buffer length in bytes. The procedure uses a number of bytes starting at address AREA which is as follows:

```
     2* SHARES * (7 + (LENGTH + 1) //2 )  bytes.
```

This part of AREA is structured as follows:



All sharestates are set to free. The file variables in the zone are set as follows:

```
    Z.zlength    =  LENGTH
    Z.zsharel    =  LENGTH
    Z.zused     is  initialized.
```

No checking is done whether the storage area provided by AREA is large enough, and consequently some storage locations may be overwritten if there is not room enough.

Note that an additional number of message buffers equal to (SHARES - 1) may have to be created, e.g. by CREATE MESSBUFS (3.5.).

4.2. Set Conversion Table Address.

Declaration:

```
procedure CHANGETABLE    (file    F;
                          const   IDENT:    string(6)  );
codebody;
```

The parameter IDENT contains a program name, unused bytes filled with zero. The procedure searches the MUS program chain for a program item with name IDENT. If the name is found, the program item is assumed to contain a conversion table, which address is set up in F.ZCONV. If the name is not found, F.ZCONV is cleared to zero.

The format of the conversion table module is:

In MUSIL:

```
     const
        TABLE =  # ... #   ;  ! conversion table!
     begin
        ...  ! statement part may be empty !
     end
```

In assembler:

```
     <program head>         ;
TSTART: LDA  2,CUR          ; empty statement part
        STOPPROCESS         ;
        JMP  .-1            ;
        0                   ; 2 zeroes
        0                   ;
        <table>             ; conversion table
        <process descriptor>;
```

4.3. Get State of Share.

Declaration;

```
procedure ZONESTATE     (file    Z;
                          var     SHARESTATE:   integer  );
codebody P0052;
```

The procedure returns the sharestate given for the share

Z.zused.snext

in the parameter SHARESTATE as follows

```
SHARESTATE = 0   share is free
           > 0   share is in use, not yet processed by the
                 driver.
           < 0   share is in use, but has been processed by
                 the driver.
```

If Z is single buffered, the share in question is the singular share. If Z is multiple buffered, the share is the next to be used. An I/O procedure with a waiting point, e.g. wait transfer, will not be delayed if SHARESTATE < 0 or = 0. Procedures waiting for all shares, e.g. waitzone, setposition or close, may be delayed.

The procedure TESTZONE (P0083) is a simpler version than ZONESTATE.

5. MUSIL UTILITIES. 5.

These procedures extend the standard MUSIL set of procedure and operators.

5.1. Arithmetic and Bit Manipulating Procedures. 5.1.

5.1.1. Double Precision Integer Arithmetic. 5.1.1.

Declarations:

```
procedure DADD          (var    RESULT:   string(4);
                         const  OP1:      string(4);
                         const  OP2:      string(4)  );
codebody P0167;

procedure DSUB          (var    RESULT:   string(4);
                         const  OP1:      string(4);
                         const  OP2:      string(4)  );
codebody P0168;

procedure DMULT         (var    RESULT:   string(4);
                         const  OP1:      string(4);
                         const  OP2:      string(4)  );
codebody P0169;

procedure DDIV          (var    RESULT:   string(4);
                         var    REM:      string(4);
                         const  OP1:      string(4);
                         const  OP2:      string(4)  );
codebody P0170;
```

The procedures work on a common representation of doubleword integers. A doubleword integer is represented by a 4 byte string, in two's complement, which may be declared like this:

```
record
     MOST:    integer;
     LEAST:   integer
end;
```

A zero is represented as MOST = LEAST = 0. No checking or indication is given on overflow.

Example:

Three variables,declared like this:

```
type DI =   record
                MOST,LEAST:  integer
            end;

var  DA, DB, DC, DSUM, DR:  DI;
```

The doubleword average of DA and DB may be computed by:

```
            DC.MOST:=  0;
            DC.LEAST:= 2;    ! the constant 2 !

            DADD (DSUM, DA, DB);
            DDIV (DSUM, DR, DSUM, DC);
```

The result is in DSUM.

If D2 is a constant:

```
     const  D2 =  # 0 0 0 2 # ;
```

then DC is set like DC:=D2 ;

5.1.2. Add Bits. 5.1.2.

Declaration:

```
procedure LOGOR          (var      RESULT:    integer;
                          const    NEWBITS:   integer   );
codebody P0081;
```

The procedure performs

```
     RESULT := RESULT or NEWBITS
```

the 'or' being a logical 16 bit parallel 'or' given by the table

```
     0 + 0 = 0
     0 + 1 = 1
     1 + 0 = 1
     1 + 1 = 1
```

5.1.3. Subtract Bits. 5.1.3.

Declaration:

```
procedure SUBBITS        (var      RESULT:    integer;
                          const    BITS:      integer   );
codebody P0184;
```

The procedure removes the bits given in BITS from RESULT, if present:

```
     RESULT:= RESULT and (8'177777 - BITS),
```

i.e. after the table

old RESULT bit		BITS bit		new RESULT bit
0	-	0	=	0
0	-	1	=	0
1	-	0	=	1
1	-	1	=	0

5.2. String Manipulation.

5.2.1. Data Movement.

Declarations:

```
procedure AMOVE          (const   FROMADDR:  integer;
                          const   TOADDR:    integer;
                          const   COUNT:     integer   );
codebody P0120;

procedure MOVIN          (const   FROMSTR:   string(1);
                          const   PROMINDX:  integer;
                          const   TOADDR:    integer;
                          const   COUNT:     integer   );
codebody P0160;

procedure MOVOUT         (const   FROMADDR:  integer;
                          var     TOSTR:     string(1);
                          const   TOINDX:    integer;
                          const   COUNT:     integer   );
codebody P0161;
```

These procedures form an extension to the MUSIL standard MOVE.

The parameters designated as FROMADDR and TOADDR are absolute byte storage addresses. The number of bytes moved is given by COUNT. The parameters FROMSTR and TOSTR are MUSIL strings, but the datamovement is done at the byte with displacement FROMINDX and TOINDX, respectively. Displacement zero indicates first byte. The move is done by the MUS utility MOVE.

5.2.2. Insertion and Extraction.

Declarations:

```
procedure IEXTRACT      (var    RESULT:    integer;
                         const  FROMSTR:   string(1);
                         const  FROMINDX:  integer  );
codebody P0123;

procedure IINSERT       (const  VALUE:     integer;
                         var    TOSTR:     string(1);
                         const  TOINDX:    integer  );
codebody P0124;

procedure INVALUE       (const  VALUE:     integer;
                         const  TOADDR:    integer;
                         const  ATYPE:     integer  );
codebody P0121;

procedure OUTVALUE      (var    RESULT:    integer;
                         const  FROMADDR:  integer;
                         const  ATYPE:     integer  );
codebody P0122;

procedure SETINTEGERS   (const  VALUE1:    integer;
                         const  VALUE2:    integer;
                         var    TOSTR:     string(1);
                         const  TOINDX:    integer  );
codebody P0182;
```

These procedures form an extension to the MUSIL byte and word operators.

Parameters TOSTR and FROMSTR are MUSIL strings, and TOINDX and FROMINDX displacements, the value zero indicating first byte. FROMADDR and TOADDR are absolute storage byte addresses.

- The procedures IEXTRACT and IINSERT extract an integer field from a string, respectively inserts an integer into a string. The displacements indicate the place of the most significant 8 bits, and the least significant bits are placed in the following byte.

The procedures INVALUE and OUTVALUE work on one or two bytes according to the value of ATYPE:

ATYPE	INVALUE	OUTVALUE
1 (byte)	insert VALUE (8:15)	fetch one byte
2 (word)	insert VALUE (0:15) in two bytes	fetch two bytes

The procedure SETINTEGERS inserts two integers into a string:

```
TOSTR(TOINDX;TOINDX+1) :=   VALUE1
TOSTR(TOINDX+2,TOINDX+3) := VALUE2
```

5.3. Data Conversion.

5.3.1. Move with Data Conversion.

Declarations:

```
procedure ACONVERT      (const  FROMADDR:  integer;
                         const  TOADDR:    integer;
                         const  TABLEADDR: integer;
                         const  COUNT:     integer  );
codebody P0131;

procedure CONVIN        (var    FROMSTR:   string(4);
                         const  FROMINDX:  integer;
                         const  TOADDR:    integer;
                         const  COUNT:     integer;
                         const  TABLEADDR: integer  );
codebody;
```

```
procedure CONVOUT        (const   FROMADDR:  integer;
                          var     TOSTR:     string(1);
                          const   TOINDX:    integer;
                          const   COUNT:     integer;
                          const   TABLEADDR: integer  );
codebody;
```

These procedures work like AMOVE, MOVIN and MOVOUT (see 5.2.1.), except that they convert the bytes moved by means of the table specified by the absolute byte storage address TABLEADDR.

5.3.2. Binary to Octal. 5.3.2.

Declaration:

```
procedure BINOCT         (const   NUMBER:    integer;
                          var     TEXT:      string(6)  );
codebody P0087;
```

The 16 bit integer NUMBER is converted to an ASCII text, consisting of 6 octal digits.

Example:

The number 600 (decimal) is converted to the text '001130'.

5.3.3. Binary to EBCDIC. 5.3.3.

Declaration:

```
procedure BINTOEBCDIC    (var     LASTCHAR:  string(1);
                          const   NUMBER:    integer   );
codebody P0017;
```

- The integer NUMBER is an unsigned 16 bit value. It is converted to a string of EBCDIC digits, with the last digit placed in LASTCHAR. At least one digit is produced, at most 5. The receiving string may be declared like

```
EBCDICNUM:   record
                  digits:     string(5);
                  zero:       string(1);
                  lastchar:   string(1) from 5
             end;
```

EBCDICNUM may be filled with spaces (64), zeroes (240) or stars before the procedure is called.

5.3.4. Convert to Hexadecimal.

Declaration:

```
procedure HEX          (var     SOURCE:    string(1);
                        const   FROMINDX:  integer;
                        var     RESULT:    string(1);
                        const   COUNT:     integer    );
codebody P0018;
```

The procedure converts an input string SOURCE of 8-bit bytes into an output string RESULT consisting of 2 hexadecimal digits for each byte, coded in EBCDIC. The 2 hexadecimal digits are the most significant 4 bits (zone) followed by the 4 least significant bits (numeric). The parameters COUNT and FROMINDX determines the portion of SOURCE converted.

SOURCE(FROMINDX) ... SOURCE(FROMINDX+COUNT-1).

The string RESULT should be at least 2* COUNT bytes long.

Example: a byte of value 8'101 (ASCII A) is converted into two EBCDIC bytes with values 244 and 241 (Hex 41).

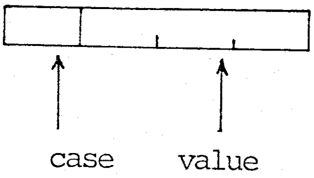
5.3.5. Convert to FLEXOWRITER Code.

Declaration:

```
procedure FLEXO          (const   FROMSTR:   string(1);
                           var     COUNT:     integer;
                           var     TOSTR:     string(1);
                           const   TABLE:     string(1);
                           var     CASE:      integer    );
codebody P0118;
```

The procedure converts the bytes in FROMSTR into FLEXOWRITER code by means of TABLE. The result TOSTR contains appropriate case shift characters.

The conversion table has the following format. For each input value the value of byte TABLE (input char) is



00	0	skip character
00	1	skip character and following.
00	>1	as case = 10
01	value	lower case character
10	value	upper case character
11	value	case independent character.

All 8 bits are put into TOSTR.
The parameter CASE contains current case, 64 meaning lower, 128 meaning upper. It is as well a call parameter as a return parameter. The parameter COUNT contains at call the number of bytes in FROMSTR, at return the actual number of bytes put into TOSTR.

5.4. Insert Duplicate Bytes.

Declarations:

```
procedure AFILL          (const   BYTEVALUE: integer;
                          const   TOADDR:    integer;
                          const   COUNT:     integer    );
codebody P0119;

procedure FILL           (const   BYTEVALUE: integer;
                          var     TOSTR:     string(1);
                          const   TOINDX:    integer;
                          const   COUNT:     integer    );
codebody P0082;
```

The procedures insert the value given in BYTEVALUE (extract 8) in the destination string in COUNT consecutive bytes. FILL inserts in TOSTR(TOINDX) ... TOSTR(TOINDX+COUNT-1).
AFILL uses the absolute byte storage address TOADDR.

5.5. Find Addresses.

5.5.1. Internal String Address.

Declaration:

```
procedure TAKEADDRESS    (const   STR:       string(1);
                          var     ADDR:      integer    );
codebody P0159;
```

The absolute byte storage address of the first byte of STR is returned in ADDR.

5.5.2. External Table Address. 5.5.2.

Declaration:

```
procedure GETTABLEADDRESS   (var    ADDR:    integer;
                             const  IDENT:   string(6)  );
codebody P0125;
```

Performs the same functions ad CHANGETABLE (4.2), except that the conversion table address in returned in ADDR, instead of in F.ZCONV.

5.6. Array Simulation. 5.6.

5.6.1. Integer Arrays. 5.6.1.

Declarations:

```
procedure LOAD          (var    BASE:    integer;
                         const  INDEX:   integer    );
codebody;

procedure STORE         (var    BASE:    integer;
                         cosnt  INDEX:   integer    );
codebody;
```

The procedures make use of a pseudoarray:

```
var        BASE:  integer;
           AREA:  string(L);
```

The length L is two times the number of array elements. The BASE serves two purposes. It contains the array element value being worked on, and it is used to address the array structure. The AREA must be declared immediately after BASE and contains the array elements.

- The INDEX determines the array element in question, the value being between 0 and L/2-1. No checking is done on this. LOAD transfers the element with index INDEX to BASE. STORE is the opposite operation. The calls LOAD(A,-1) and STORE(A,-1) have no effect.

5.6.2. General Arrays.

Declarations:

```
procedure INITAREA       (var    AREA:      integer;
                          var    VARS:      string(1)   );
codebody P0074;

procedure SWAPVARS       (var    AREA:      integer;
                          const  NEWINDEX:  integer     );
codebody P0075;
```

The procedures use a pseudoarray with the following general structure:

```
var
      AREA :   integer;     ! 3 words header containing          !
      HEAD :   string(4);   ! current index, length and address  !
                            ! of element no. 0                   !

      ! here declaration of user variables !

      VARS :   string(L);   ! actual elements                    !
```

The length L of the string containing the elements (here VARS) should be computed as the size in bytes of the user variables declared between HEAD and VARS, multiplied by the maximum number of elements (called N).

The following table summarizes the length in words for various data structures. The length in bytes of the user area will be two times the sum of all length's (in words).

Data Structure	Length in words.
a. integer	1
b. string(L1)	(L1+1)/2

c. record
V1 : T1;
V2,V3: T2;
:
Vj : Tj;
Vk : Tk from Pk;
.
:
end;
	Each subrecord has a length in bytes. Subrecords without FROM keyword is put immediately after the previous.
	Length in words is half the total extent of the record.

d. file ...,...,N1,L1,... of ....; ! length = L2 !	Format F,FB: L2*p = L1, some p≥1. Length in words: 26+N1*(7+(L1+1)//2)

The procedure INITAREA initializes the head of the area. Current index is set to 1, meaning that the first array element is assumed to be available to the user.

The procedure SWAPVARS compares current index with NEWINDEX. If they are different, the variables in the user area is moved to the part of VARS where they belong, the variables belonging to NEWINDEX are fetched, and current index is updated. No checking is done whether NEWINDEX is between 1 and N.

5.7. Queue Administration.

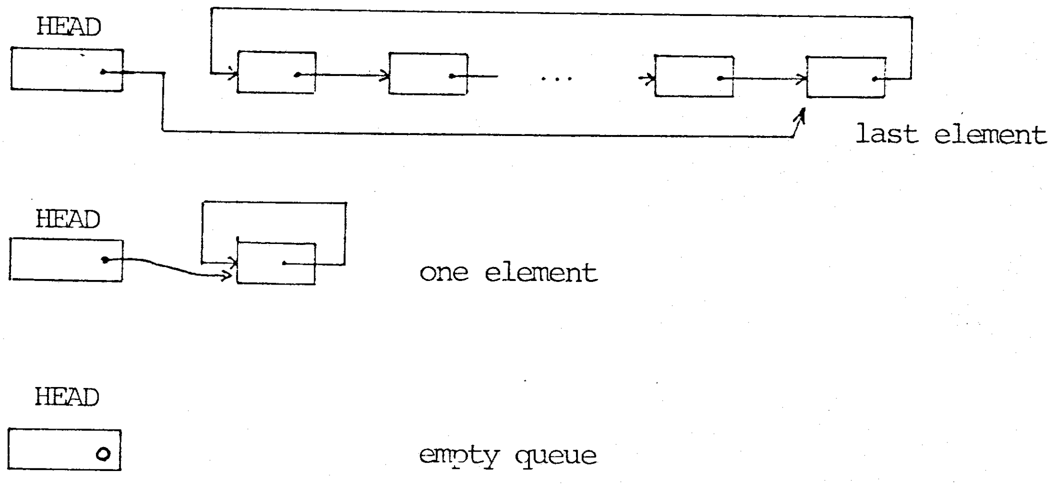
Declarations:

```
procedure CHAININ      (var   OPADR:   integer;
                        var   HEAD:    integer    );
codebody P0076;

procedure CHAINOUT     (var   HEAD:    integer;
                        var   OPADR:   integer    );
codebody P0077;

procedure EXAMINE      (var   OPADR:   integer    );
codebody P0078;
```

The queues are first in/first out queues structured like this:



HEAD and link fields are one word long. Their values are absolute storage addresses.

As the procedures are related to the Coroutine Monitor procedures and data structures, the description of the 3 procedures are found in the MUSIL Coroutine Programmer's Reference (see 5.8.).

5.8. Coroutine Procedures.

These procedures with associated data structures are described in:

Extended RC3600 Coroutine Monitor
MUSIL Programmer's Manual.

The procedure set includes

P.....	NAME
72	TESTPOINT
73	RESETSTACK
79	CHANGEMASK
80	CDELAY
88	INITCOSYS
89	DEFCOROUT
90	SETUSEREXIT
91	INITGENSEM
92	WAITGENERAL
93	SIGNAL GENERAL
94	RETURN ANSWER
95	CSENDMESSAGE
96	SAVELINK
97	RETURN
98	RELEASE ANSWER
126	PASS
127	WAITSEM
128	SIGNAL

5.9. Special Procedures.

The following procedures have special functions, but are of general use in specific environments.

P....	NAME	FUNCTION
-	INITSCDPROG	Used in connection with the SCD - driver series.
-	SETSCDPROG	
152	DUMMY	Used when several programs shares a codeprocedure library, to save space.
153	INITCODEP	
157	FIX	Used to place buffer areas in MUSIL initialization code.
158	ALLOCATE	

See the listings for description.

6. LIST OF CODEPROCEDURES.

6.1. Codeprocedures without 'P....' Names.

NAME	RCSL 43-GL.... (BIN.TAPE)	DESCRIPTION
AANSWER	607	(2.1.2.)
CHANGETABLE	1519	(4.2.)
CONVIN	613	(5.3.1.)
CONVOUT	616	(5.3.1.)
FINDPROCESS	625	(2.3.5.)
INITSCDPROG	1251	(5.9.)
LOAD	631	(5.6.1.)
OPERATOR	?	(3.3.2.)
SETSCDPROG	?	(5.9.)
SMESSAGE	649	(2.1.1.)
SPRIO	?	(3.3.3.)
STORE	655	(5.6.1.)
TESTMESSAGE	664	(2.1.3.)

6.2. Codeprocedures with 'P....' Name.

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
1		GETTIME	Replaced by P0149.
2		GETTIME	Special version of P0001.
3		GETDATE	Replaced by P0150.
4		GETDATE	Special version of P0003.
5		GETDATE	Replaced by P0151.
6		GETDATE	Special version of P0005.
7	1406	SEARCHITEM	(2.3.3.)
8		SCAN RECORD	Special purpose.
9		ADD DOUBLE	-
10		GETNAME	Outdated.
11	375	COMPUTE DATE	(3.1.2.)
12		COMPARE	Special purpose.
13		GETADDRESS	See P0159.
14		RESERVE STAT	Special purpose.
15		MOVE BUFFER PART	Outdated.
16		INREC	Special purpose.
17	393	BIN. TO EBCDIC	(5.3.3.)
18	396	HEX. OUTPUT	(5.3.4.)
19		?	
20		CHECK NUMERIC	Special purpose.
21		GET CARD INFO.	Special purpose.
22		CHANGETABLE	-
23	1409	DELAY	(2.3.1.)
24		MEASURE FREE CPU	Special purpose.
25		?	
26		?	
27		EXPAND	Special purpose.
28		SET ZKIND	Outdated.
29		PACK	Special purpose.
30		MT/PTR to LP CONV.	-

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
31		PTR to MT CONV.	Special purpose
32*		ASEARCH	-
33*		SETTAB	-
34*		SETNOTAB	-
35*		SETTAB	-
36		SEARCH	-
37		SPECIAL GETREC	-
38		CHANGE PRINTTABLE	-
39		WAITMESSAGE	See P0156.
40	2618	SENDANSWER	(2.2.3.)
41		COMPRESS	Special purpose.
42		DECOMPRESS	-
43		DECOMPRESS	-
44*		BUFRETURN	See P0045.
45		BUFRETURN	Special purpose.
46*		GETEVENT	See P0047.
47	2135	GETEVENT	(2.2.1.)
48	2177	STOPPROCESS	(2.3.2.)
49*		SCAN	Special purpose.
50		ADD/SUB STRING	-
51*		CASE	-
52	1929	ZONESTATE	(4.3.)
53		REAL TIME COUNTER	Special purpose.
54	2350	CREATE MESS.BUF.S	(3.5.)
55	2403	GETCURNAME	(3.3.1.)
56		SETSHARE	Replaced by P0155.
57		DEPACK	Special purpose.
58		COMPARE	-
59		FLEX.DISC Handler	-
60*		RANDOM (Special)	-

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
61*		RANDOM (Special)	Special purpose.
62		VALIDERING (1)	-
63		- (2)	-
64*		EXPAND	See P0065.
65		EXPAND	Special purpose.
66		STATCOUNT	-
67		AJOURSTAT	-
68		GETPROMILLE	-
69		SIM. VFU	-
70		VFUTIME	-
71		CONSTRING	Special prupose.
72	3506	TESTPOINT	Note (1)
73	3509	RESETSTACK	Note (1)
74	3512	INITAREA	(5.6.2.)
75	3515	SWAPVARS	(5.6.2.)
76	3518	CHAININ	(5.7.)
77	3521	CHAINOUT	(5.7.)
78	3524	EXAMINE	(5.7.)
79	3527	CHANGEIDENT	Note (1)
80	4058	CDELAY	Note (1)
81	637	LOGOR	(5.1.2.)
82	622	FILL	(5.4.)
83	667	TESTZONE	Replaced by P0052.
84	3134	FINIS	Note (2)
85	3137	GETPARAMS	Note (2)
86	3275	CONNECTFILE	Note (2)
87	3301	BINOCT	(5.3.2.)
88	3304	INITCOSYS	Note (1)
89	3307	DEFCOROUT	Note (1)
90	3310	SETUSEREXIT	Note (1)

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
91	3313	INITGENSEM	Note (1)
92	3316	WAITGENERAL	Note (1)
93	3319	SIGNAL GENERAL	Note (1)
94	3322	RETURN ANSWER	Note (1)
95	3325	CSENDMESSAGE	Note (1)
96	3328	SAVELINK	Note (1)
97	3331	RETURN	Note (1)
98	3334	RELEASE ANSWER	Note (1)
99		INITTEST	Special purpose.
100		GETTREG	-
101		FREETREC	Special purpose.
102		SETSHARE	Replaced by P0155.
103		SWOP	Special purpose.
104		ABSSTRING	-
105		GETSLICE	-
106		SENDWRITE	-
107		WAITWRITE	-
108		READ	-
109		COMPARE	-
110		STOPTEST	-
111		CONVERT/CHECK STRING	Special purpose.
112		SIEMENS DECOMPRESS	-
113		SENSE DISC	-
114		SCAN STRING	-
115		READ BIN.CARD	-
116		WRITE BIN.CARD	-
117	3933	GET CORE ITEM	(2.3.4.)
118	4030	CONVERT TO FLEXO.	(5.3.5.)
119	5674	AFILL	(5.4.)
120	5677	AMOVE	(5.2.1.)

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
121	5680	INVALUE	(5.2.2.)
122	5683	OUTVALUE	(5.2.2.)
123	5686	IEXTRACT	(5.2.2.)
124	5689	IINSERT	(5.2.2.)
125	5692	GET TABLEADDRESS	(5.5.2.)
126	?	PASS	Note (1)
127	?	WAITSEM	Note (1)
128	?	SIGNAL	Note (1)
129		BCC 93	
130		SETSHARE	Replaced by P0155.
131	5695	ACONVERT	(5.3.1.)
132	4158	LOOKUP CATW	(3.4.1.)
133		SWAPNAMES	Special purpose.
134		ALTERNATE	Special purpose.
135		FLOATING ZERO	-
136		XCOUNT	-
137		RSTCP	-
138		CONNECT EMULATOR	-
139		SETSHARES	Replaced by P0155.
140		DT OPEN	Special purpose.
141		DT CLOS	Special purpose.
142		DT MAN	-
143		SPLITUP	-
144		PUTENTRY	-
145		SEARCH ENTRY	-
146		DEPACK	-
147		COMPRIM	-
148		COMPARE	-
149	4928	GETTIME	(3.1.3.)
150	4931	GETDATE	(3.1.1.)

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
151	4934	GETDATE	(3.1.1.)
152	5005	DUMMY	(5.9.)
153	5008	INITCODEP	(5.9.)
154	5040	SET ENTRY	(3.4.2.)
155	5161	INITZONE	(4.1.)
156	5186	WAIT MESSAGE	(2.2.2.)
157	5347	FIX	(5.9.)
158	5350	ALLOCATE	(5.9.)
159	5698	TAKEADDRESS	(5.5.1.)
160	5701	MOVIN	(5.2.1.)
161	5704	MOVOUT	(5.2.1.)
162		EXTA-RELBITS	Special purpose.
163		GOABS	-
164	5895	SMESS	(2.1.1.)
165	5898	STARTPROCESS	(2.3.2.)
166		EXPAND (SHELL)	Special purpose.
167	6028	DOUBLE ADD	(5.1.1.)
168	6031	DOUBLE SUBTRACT	(5.1.1.)
169	6034	DOUBLE MULTIPLY	(5.1.1.)
170	6090	DOUBLE DIVIDE	(5.1.1.)
171		XMIT IMAGES	Special purpose.
172		REC IMAGES	-
173		ACCESS	Note (3)
174		ALLACCESS	Note (3)
175		GET COMMAND	Note (3)
176		DELAY	Note (3)
177		FITEM	Note (3)
178		GET NEXT ITEM	Note (3)
179		GET PARAMETER	Note (3)
180		RETURN	Note (3)

P....	RCSL 43-GL.... (BIN.TAPE)	NAME	DESCRIPTION
181	6481	COMPUTE MINUTES	(3.1.4.)
182	6590	SET INTEGERS	(5.2.2.)
183		INCREMENT	Special purpose.
184	6596	SUBBITS	(5.1.3.)
185		COREDUMP	Special purpose.
186		GET RECORD	-
187		INC. INTEGER	-
188		PLAYC	-
189	÷	NODC	-

A star (*) means that the name (.TITL) is not P....

NOTES:

(1) Coroutine procedure. See separate manual. (5.8.).

(2) DOMUS procedure. See separate manual.

(3) Data Entry, release 2. See separate description.