1.4 Procedure Division

1.4.1 *Introduction*

The procedure division contains the instructions which direct the input, processing and output of data. It also contains some declarations which must be used in conjunction with certain instructions. The use of directives in the procedure division is discussed in Section 1.2. The general layout of the procedure division is shown below.

The ENTRY and EXT directives (if present) must be the first statements in the procedure division. Either ENTRY or EXT may be written first. The EQU directive may occur anywhere in the procedure division, after the ENTRY and EXT directives. The procedure division continues with the remaining instructions and declarations written in a sequence dictated by the programmer.

Subroutines (enclosed in PROC and PEND directives) may appear anywhere in the remainder of the procedure division. It is often desirable to make the whole of one module a subroutine. This is achieved simply by making the PROC directive the first statement after the ENTRY/EXT/EQU cluster and by making the PEND directive the last statement before the END directive.

```
[ENTRY etc]                           – Entry point directive(s)
[EXT etc]                             – External reference directive(s)
[EQU etc]                             – Equate directive(s)
Instructions/Declarations in any sequence
        .
        .
        .
   PROC                               – Subroutine start directive
   Instructions/Declarations in any sequence
        .
        .
        .
   PEND                               – Subroutine end directive
        .
        .
        .
        .
```

1.4.2 *Instructions*

1.4.2.1 *General*

The general format of an instruction is:

[statement-identifier] ␣ instruction-mnemonic ␣ [operand] [,operand] . . . .

The "instruction-mnemonic" specifies the basic operation to be performed by the instruction. This mnemonic may be followed by one or more "operands". These operands have a different significance for each instruction. The operands of a particular instruction are often referred to as operand-1 , operand-2 etc. The leftmost operand in an instruction is counted as one.

1.4.2.2 *Arithmetic Instructions*

These instructions are:

Mnemonics	Significance
ADD	Add
CMP	Compare
DIV	Divide
DVR	Divide rounded
MOVE	Move (conversions)
MUL	Multiply
SUB	Subtract

With the exception of MOVE all arithmetic instructions must operate on data items of the same type. That is, the operands must both be binary, decimal or string.

The MOVE and CMP instructions may operate on binary, decimal or string data items. The remaining arithmetic instructions operate on binary or decimal data itemr only.

The format of an arithmetic instruction consists of an operation code followed by two operands.

1.4.2.3 *Branch Instructions*

These instructions are:

Mnemonics	Significance
CB	Compare and branch
IB	Indexed branch
LB	Long branch
SB	Short branch (within 255 bytes)
TB	Test and branch (Boolean data-items)

All branch instructions, except IB, contain a condition mask. This is an integer ranging from 0 to 7 inclusive. If the condition mask corresponds with the contents of the condition register the branch instruction is obeyed. Otherwise the instruction following the branch is executed.

In some branch instructions the condition mask is included in the mnemonic as e.g. branch on equal (BE), branch on OK (BOK) or branch on error (BERR). The translator will decide if this is going to be a short branch or long branch.

Mnemonics	Significance
B	Branch
BBEOD	Branch on begin/end device
BE	Branch on equal
BEOF	Branch on end of file
BERR	Branch on error
BG	Branch on greater
BL	Branch on less
BN	Branch on negative
BNE	Branch on not equal

Mnemonics	Significance
BNEOF	Branch on no end of file
BNERR	Branch on no error
BNG	Branch on not greater
BNL	Branch on not less
BNN	Branch on not negative
BNOK	Branch on not OK
BNP	Branch on not positive
BNZ	Branch on not zero
BOFL	Branch on overflow
BOK	Branch on OK
BP	Branch on positive
BZ	Branch on zero
CBE	Compare and branch on equal
CBG	Compare and branch on greater
CBL	Compare and branch on less
CBNE	Compare and branch on not equal
CBNG	Compare and branch on not greater
CBNL	Compare and branch on not less
TBF	Test and branch on false
TBT	Test and branch on true

The condition register is a two bit register which is automatically set during the execution of certain instructions. It may contain an abbreviated status code, the previous value of a boolean data item or the result of a compare instruction. The condition register is used by the following instructions:

Contents of condition register:	Instructions which set the condition register:
Abbreviated status code	ADD, INSRT, MATCH, SUB, USE, MUL, DIV, DVR and all I/O instructions
Previous value	All logical instructions
Result of comparison	CMP, CB, TB and TEST

The CB and TB instructions are the only branch instructions which actually set the condition register.

The function of branch instructions is to control the instruction execution sequence by updating the program pointer (PP). During the execution of a CREDIT program the program pointer holds the address of the next interpretive instruction to be executed.

The SB, CB and TB may branch forwards or backwards up to 255 bytes. The LB and IB instructions may branch forwards or backwards any number of bytes within addressable memory.

In the virtual memory system, each segment will contain a long branch table. Each table may contain up to 255 entries.

1.4.2.4 *Input/Output Instructions*

These instructions are:

Mnemonics	Significance
ABORT	Abort I/O operation
ASSIGN	Assign a data file

Mnemonics	Significance
ABORT	Abort I/O operation
ASSIGN	Assign a data file
DSC0	Data set control zero
DSC1	Data set control one
DSC2	Data set control two
EDWRT	Edit and write
IASSIGN	Assign an index file
IINS	Indexed insert
IREAD	Indexed random read
IWRITE	Indexed rewrite
KI	Keyboard input
MWAIT	Multiple wait
NKI	Numeric keyboard input
READ	Read
RREAD	Random read
RWRITE	Random write
TESTIO	Test completion I/O
WRITE	Write
WAIT	Wait
XSTAT	Extended status transfer

I/O instructions operate upon data sets. These are referred to by using the data set identifiers included in the DSET declarations.

Unless the "no wait" option is specified in an I/O instruction, execution of the task will be suspended during each I/O operation and will not be re-started until the I/O operation is complete. The "no wait" option is specified by including .NW in the I/O instruction.

This results in the I/O being started and the task being put directly into the dispatcher queue. While I/O is being performed, the task may gain control. When the task reaches a stage at which further processing is impossible until the I/O is completed, it can request that execution be suspended by executing the WAIT instruction.

Unless the "no echo" option is specified in a keyboard input instruction (KI, NKI), the input data will be echoed on the associated echo device. The echo device associated with each keyboard is specified when the TOSS Monitor is generated. It may be a Visual Display Unit, a Plasma Display Unit, a Numeric and Signal Display Unit or a General Terminal Printer. The "no echo" option is specified by including .NE in the instruction.

If the EDWRT instruction is used the associated DSET declaration must specify a buffer length. This is because this instruction edits directly into a buffer specified by the CREDIT Translator. The keyboard input instructions (KI, NKI) and the READ and WRITE instructions use buffers specified by the CREDIT programmer in the appropriate work blocks.

During the I/O operation a device dependent status code is generated (known as the extended status code). The TOSS Monitor then generates an abbreviated status code which it places in the condition register. This status code summarises the conditions indicated by the extended status code. The abbreviated status code is generated in the following way.

The extended status code is compared with the mask X'E8DF'. If there are any corresponding bits set to '1' in both words then the value 2 (error) is placed in the condition register. If none of the '1' bits matches then the extended status code is compared with the mask X'0420'. If any '1' bits match then the value 3 (begin or end of device) is placed in the condition register. If there is still no match then the extended status register is compared with the mask X'1000'. If the '1' bit matches then the value 1 (end of file) is placed in the condition register.

Since the sum of all the masks is 'FCFF', two bits are not checked. These may be checked by the CREDIT programmer if necessary.

An extended status code may be obtained by the CREDIT program via the XSTAT instruction. The extended status code for each type of data set is described in appendix 2.

Data set control (rewind of tape, grasp action of Teller Terminal, switching indicator lights etc) is achieved by the data set control instructions (DSC0, DSC1, DSC2).

1.4.2.5 *Logical Instructions*

These instructions are:

Mnemonic	Significance
CLEAR	Clear (Reset)
INV	Invert
SET	Set
TEST	Test boolean

Logical instructions operate upon boolean data items. At the completion of a logical instruction the condition register is set at the *previous* value of the boolean data item.

Each logical instruction occupies two bytes of core. The first byte contains the operation code and the second byte is a reference to the boolean data item.

1.4.2.6 *Scheduling Instructions*

These instructions are:

Mnemonic	Significance
ACTV	Activate an other task
EXIT	Terminate a task
DELAY	Delay task execution
GETID	Get task identifier
PAUSE	Inhibit a task
RSTRT	Restart paused task
SWITCH	Switch control to another task

Scheduling instructions are used to activate or restart a task in a different terminal class (ACTV, RSTRT) or to pause or terminate the current task (PAUSE, EXIT).

All scheduling is done by the TOSS Monitor. The task identifier of each active task is held in a "dispatcher" queue. This is a "first in first out" queue of tasks awaiting execution. When the currently executed task cannot proceed, for any reason, the TOSS Monitor hands control to the next task in the dispatcher queue.

If an executing task performs an EXIT instruction, the TOSS Monitor will deactivate the task. That is, execution will be terminated and all records of the task in the TOSS Monitor will be deleted. Such a task may be re-activated by a task in the same or another terminal class which performs an ACTV instruction for the de-activated task. The task will then be initialised and reinserted in the dispatcher queue.

If an executing task performs a PAUSE instruction, the TOSS Monitor will place the task in a "pending" state. That is, execution of the task will cease and its task identifier will not be entered in the dispatcher queue. However, all registers will be saved. Such a task may be restarted by a task in the same or another terminal class which performs a RSTRT instruction for the pending task. The task will then be reinserted in the dispatcher queue.

The difference between the PAUSE and EXIT instructions is that after a PAUSE the task remains active (and therefore cannot be activated by an ACTV instruction), whereas after an EXIT instruction the task becomes inactive.

1.4.2.7 *Storage control instruction*

These instructions are:

Mnemonic	Significance
USE	Attach User of Swappable work block
UNUSE	Detach User or Swappable work block

With the USE instruction a user work block or swappable work block can be attached to the current task. A swappable work block will be loaded into main memory, from disc. Execution of the UNUSE instruction results in a detaching of a user work block or swappable work block from the current task. The swappable work block will be rewritten on disk.

1.4.2.8 *String Instructions*

These instructions are:

Mnemonic	Significance
COPY	Copy
DLETE	Delete
EDIT	Edit buffer
EDSUB	Edit substring
INSRT	Insert
MATCH	Match
XCOPY	Extended copy

The string instructions are used to manipulate string data items. The COPY and XCOPY instructions may also be used with decimal data items.

String instructions occupy from three to seven bytes of core.

1.4.2.9 *Subroutine Control Instructions*

These instructions are:

Mnemonic	Significance
CALL PERF PERFI RET	Call assembler subroutine Call CREDIT subroutine Indexed perform Return from subroutine

They are used to transfer control to and from subroutines written in CREDIT or Assembler. PERFI, PERF and RET may be used with CREDIT subroutines only. CALL may be used with Assembler subroutines only.

1.4.2.10 *Format control I/O instructions*

These instructions are shown in the table below?

Instruction Mnemonic	Use
ATTFMT	Attach a format list.
DETFMT	Detach format list.
DISPLAY	Display a format list on the screen.
DUPL	Duplicate a data item.
DYKI	Input from the device, present in the FMTCTL declaration.
EDFLD	Edit input field.
ERASE	Erase on the screen.
GETABX	Get current input field number.
GETCTL	Get control value (MINL, MAXL etc.)
GETFLD	Get field makes input field current.
PRINT	Print format list on output device.
SETCUR	Position cursor at 1st character position of input field.
THOME, TFWD, TBWD, TRIGHT, TLEFT, TUP TDOWN TLDOWN	Tabulation.
TSTCTL	Test control flag (ME, NEOI etc.).
UPDFLD	Update field.

They operate on input fields and corresponding data items defined in a format list which is made current by the attach format instruction. Some of these instructions such as PRINT, DISPLAY, DYKI, EDFLD operate on data sets which are defined as input and output device in the format control I/O declaration (FMTCTL), in the data division (see 1.3.9).

The tabulation functions THOME, TFWD TBWD, TRIGHT, TLEFT, TUP, TDOWN and TLDOWN serve for moving the cursor to the different FKI-input fields of the current format list.

Addressing of the desired input field is always relative to the current input field An exception is THOME, which will always tabulate on the first FKI-input field of the current format list.

These tabulation functions require at least one FKI-input field to be present in the current format list.

1.4.3 *Declarations*

1.4.3.1 *Format lists*

A FRMT declaration followed by a selection of the remaining format list declarations and ending in a FMEND declaration is known as a format list. Two possibilities are available when using the format lists.

a) *The first possibility* is that lines and keyed in data are displayed or printed by using the instructions EDWRT, EDIT and WRITE.

In a format list is specified in which way an I/O buffer has to be edited. Format list declarations which may be used are:

Mnemonics	
FB, FBN, FBNN, FBNP, FBNZ, FBP, FBF, FBT, FBZ	Format branch on condition
FCOPY	Format copy
FCW	Format control word
FEOR	Format end of record
FILLR	Fill repeat
FLINK	Format link
FMEL FMELI	Format element according picture string
FMEND	Format end
FNL	Format new line
FRMT	Format start
FSL	Format start line
FTAB	Format tabulation
FTEXT	Format immediate text

An example of a format list is shown below:

Identifier	Declaration	Explanation
FORM1	FRMT	Begin format list FORM1.
	FILLR ⊔ '⊔' ,2	Spaces are inserted in columns 1 and 2 of the buffer.
	FCOPY ⊔ = C 'TERMINAL'	The characters TERMINAL are inserted in columns 3 to 10 of the buffer.
	FMEL ⊔ '99', TERM	The contents of data item TERM are edited into the buffer according to the picture 99.
	FLINK ⊔ SUBF1	The contents of format list SUBF1 are used as if they were part of this format list. Editing starts at the current position in the buffer.
	FMEND	End format list FORM1.

The above format list when used in an EDIT or EDWRT instruction would result in a buffer containing the following:

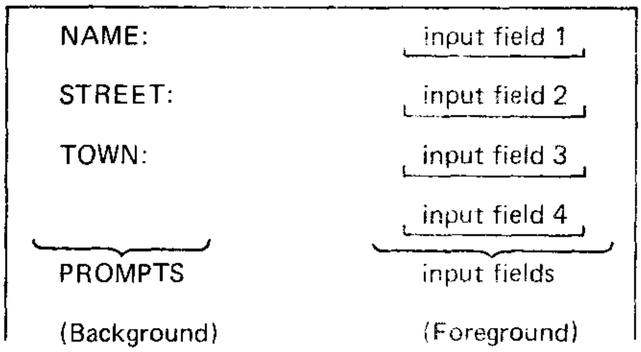
⊔ ⊔ TERMINALNN etc

where NN is the value from the data item TERM.

A pointer is maintained during the editing process which points to the buffer column into which edited data is currently being written. In the above example this pointer would have had an initial value of 1. After the FILLR declaration it would have had a value of 3. After the FMEL declaration it would have had a value of 11 and so on. If necessary, this pointer can be moved backwards or forwards through the buffer by the FTAB declaration.

As shown in the above example the FLINK declaration may be used to nest format lists and thus avoid rewriting the same sequence of declarations a number of times.

b) *The second possibility* is for one format list to describe a whole transaction layout on the screen and data which is being keyed in to be displayed on the current input field. A current input field always uses a data-item to contain the data displayed. It is now possible to display all the prompts on the screen (Background) with one instruction. These format lists may also be used by screen management (see appendix 7). With format control I/O instructions it is possible for keyed-in data to be displayed on the corresponding input field on the screen. Also data received from a disk or via a data communication line can be displayed on the desired input field on the screen.



Each input field is described, with its options, in the format list by the format list declarations format input (FINP) and format keyboard input (FKI).

As different transactions have a different layout on the screen or on the print device, each transaction can be defined complete in a format list. Only one format list (transaction) can be current for one task. A format list is made "current" by the Attach Format instruction (ATTFMT). Initially, after an attach format, none of the input fields is current. When a format list **(transaction)** is attached, it is possible to make one of the input fields current for receiving data. Only one input field may be current at a time. An input field can be made current by using one of the format control instructions such as get field (GETFLD) and the tabulation instructions THOME, TFWD, TBWD, TRIGHT, TLEFT, TUP, TDOWN and TLDOWN.

Two types of input fields are defined:

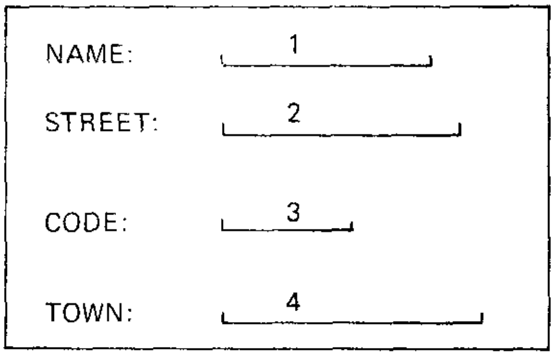
– an input field which is used to receive data from a device (except keyboard) or data item (Messages). The input field is described by the FINP declaration in the format list.

– an input field which is supposed to receive data from a keyboard. The input field is described by the FKI declaration in the format list.

These input field declarations must be followed directly by a FMEL or FCOPY format list declaration.
FMEL and FCOPY refer to decimal and string data items respectively in which the data for the input field is stored.

All input fields are referenced in the sequence as they appear in the format list.

```
FRMT
  :
FKI   .....       (1)
  :
FKI   .....       (2)

FINP  .....       (3)

FKI   .....       (4)

FMEND
```

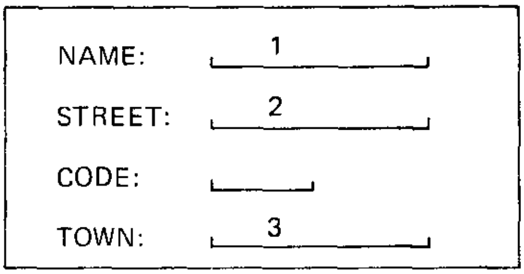


The numbering of the input fields as shown above can be selected by the user in the format control instructions e.g. GETFLD when control value 2 is specified.

To select the field sequence numbering of only FKI-fields or oniy FINP-fields the user has to specify the GETFLD instruction control value as zero for FKI-input fields and one for FINP-input fields.

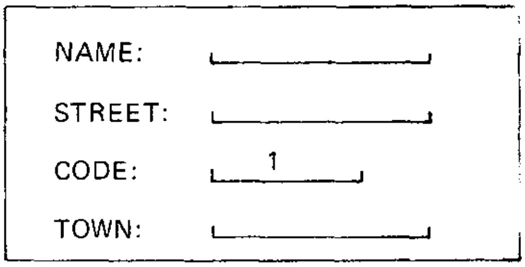
Sequence numbering of FKI-input fields only:

```
FRMT
  :
FKI     .....     (1)
  :
FKI     .....     (2)
  :
FINP    .....

FKI     .....     (3)
  :
FMEND
```



Sequence numbering for FINP-input fields only:

```
FRMT
  :
FKI     .....
  :
FKI     .....     (1)
  :
FINP    .....

FMEND
```



With the PRINT instruction a hard copy is produced on the printer (TTP or GTP) from the current format list.

When using format control I/O instructions, some rules have to be followed for using the format list.

1. The first line on the output medium must be defined in the format list by the FSL format list declaration and subsequent lines by the FNL format list declaration.
2. Data items containing variables for conditional editing in the format branch on condition declarations (FBP, FBZ etc.), may not be altered while the concerned format list is current.
3. A format list must contain at least one input field (FKI or FINP field).

However, the tabulation functions THOME, FFWD, TBWD, TRIGHT, TLEFT, TUP, TDOWN and TLDOWN require at least one FKI-field.

4. The format list declarations FCW and FEOR may only be present when immediately succeeded by a FSL or FNL format list declaration.

5. Formal parameters are not allowed in format lists which are using the format control I/O instructions.

Compulsory input fields, are fields in which data must be entered. These fields are defined in the FKI-declaration with the Must Enter (ME) bit set. When no data is entered in such an input field it can result in a condition register setting for the next executed instruction with indication "compulsory input field"

Data is not directly entered in the data-item, following the FKI-input field description. The DYKI-instruction will read data from the input device, defined in the FMTCTL declaration, and store in its own buffer. The data is echoed on the echo device, but not edited. To get the data in an edited format on the output device (e.g. screen) it has to be moved to the data-item of the current input field.

The UPDFLD instruction moves the contents of the input buffer (DYKI) to the data-item of the current input field and redisplays with editing, if so required.

When the name of the data-item of the current input field is unknown, a reserved name, :FMTITEM, is used to access this data-item. In this way data may be moved from the input buffer (DYKI) to the data item of the **current** input field.

Example: MOVE ␣ : FMTITEM, SPINPUT

(SPINPUT is the buffer present in the DYKI instruction). The other way round is also allowed, e.g. MOVE ␣ FIELD, :FMTITEM.

EDFLD instruction also uses its own buffer to update and echo it. Buffer handling is similar to the DYKI instruction.

To display the contents of the input fields belonging to data items, the DISPLAY instruction is used, which does not update the data items.

With the DUPL instruction, the contents of the data item mentioned in the DUPL option of the FKI-input field description, is moved to the data item mentioned in the DUPL instruction. When this data item happens to be a data-item of a current input field, it is not directly displayed, but must be displayed with the DISPLAY instruction.

1.4.3.2 *Key Table Declaration*

This declaration is KTAB. It is used to define a list of keyboard input termination characters. These characters are used to detect an end of message during a keyboard input operation. KTAB is used in conjunction with the keyboard input instructions KI, NKI and DYKI.

1.4.3.3 *Parameter declaration*

This declaration is PLIST. It is used to specify parameters to be passed to a subroutine when it is called with the PERFI instruction. It is recommended, not to use the CON directive since this directive does not support passing parameters (e.g. literal constants, format lists, key tables) in virtual systems or when ADRMOD=2. A PLIST directive may only be used following a PERFI instruction.

1.4.4 *Subroutine handling*

1.4.4.1 *CREDIT subroutines*

CREDIT subroutines start with a PROC directive which may be followed by up to eight formal parameters. This number depends on the addressing mode. When ADRMOD=2, two byte addressing mode, maximum 8 formal parameters are allowed. (See OPTNS directive). When ADRMOD=1, one byte addressing mode, maximum 8 bytes are available for formal parameters. In this case the maximum number of formal parameters depends on the value in LITADR. When LITADR=1111, maximum 8 formal parameters are allowed. When a formal parameter is using 2 byte addressing, selected with the LITADR option, this parameter will use 2 bytes of the maximum available 8 bytes and decreases the number of formal parameters allowed.
The number of actual parameters passed to the subroutine must be the same as the number of formal parameters in the PROC directive. Actual and formal parameters are used to pass variables to a subroutine and to store results generated by the subroutine. The variables are specified as actual parameters in a PERF instruction or PLIST directive. The format of each variable is described in a formal parameter in the PROC directive of the subroutine being called. Actual parameters are operated upon within the subroutine replacing the corresponding positional formal parameters in the instruction operands. The following list shows the types of data that can be specified as actual parameters and shows the corresponding types of formal parameter which must appear in the PROC directive.

Actual parameter	*Formal parameter*
array-identifier	identifier ( )
[index-identifier-1]	[identifier]
[index-identifier-2]	[identifier]
data-set-identifier	identifier
format-list-identifier	identifierI$identifier
format-table-identifier	identifier ( )I$identifier ( )
key-table-identifier	identifierI$identifier
literal constant	identifierI$identifier

When for at least one of the formal parameters two byte addressing is used, the PROC directive must be followed by PFRMT, PKTAB or PLIT directive, even when the other parameters are not using two byte addressing. PFRMT must always be used when a format table name is passed as parameter.
A $ sign as first symbol in the formal parameter indicates a parameter type literal constant, keytable or formatlist. When one of these parameter types is in the heading of the subroutine, without a $ sign as first symbol, the type must be specified by using the PLIT, PKTAB or PFRMT directive, also in one byte addressing mode.

Example:

```
          OPTNS     LITADR = 1111
  SUB1    PROC      FORM1, LITC, KTB1, DAT!
          PKTAB     KTB1
          PFRMT     FORM1
          PLIT      LITC

          PEND
  SUB2    PROC      FTABL ( )
          PFRMT     FTABL
```

As actual parameters may be passed:

- keytables
- format lists
- format tables
- literal constant (except type 'X')
- single data items
- one or two dimensional arrays
- data sets

When a PERF or PERFI instruction is executed the program pointer is adjusted to point to the instruction following the PERF/PERFI and is then saved on a stack. The program pointer is set to the first instruction of the subroutine.
When a RET instruction is executed the saved program pointer is restored and execution is continued at the instruction following the PERT or PERFI.

1.4.4.2 *Assembler Subroutines*

Assembler subroutines are called by the CALL instruction. It is the responsibility of the Assembler program to ensure that the program pointer is correctly stepped past any actual parameters before control is handed back to the CREDIT module. The program pointer is held in register A12. In virtual systems it is not possible to transfer parameters to assembler subroutines, except when the parameter list is picked up by the assembler routine before executing the first I/O instruction.

A number of Assembler routines are available to assist the Assembler programmer in obtaining parameter values, updating the program pointer and returning to the CREDIT module. They are I:EVA0, I:EVA1, I:EVA3; I:EVA5, I:EVA7; I:RT1 and T:FDSP.

Routine I:EVA0 is used to obtain the address of a data-item, array data-item or formal parameter

Routine I:EVA1 is used to obtain the address of a literal parameter or formal parameter

Routine I:EVA3 is used to obtain the address of a picture string or formal parameter

Routine I:EVA5 is used to obtain the address of a format list parameter or formal parameter

Routine I:EVA7 is used to obtain the address of a key table parameter or formal parameter

The return values from these routines are:

Register A3 — The data item or literal type in the right byte in bits 10 and 11.
0 indicates string, 2 indicates binary and 3 indicates decimal.

A5 — Contains the data or literal end address.

A9 — Contains the data item or literal start address.

The difference between A5 and A9 is the length of the data item, literal, picture string or format list. The contents of the registers A4, A6, A7 and A8 are not affected by these routines, and available for the user. The routines update the program pointer in the following manner

1 for data items and literals, 2 or 3 for arrays.

Routine T:FDSP may be used to obtain data set parameters. The return values from this routine are

A8 — Contains the event control block address.

A7 — Contains the wait bit in bit zero and the echo bit in bit one.

Registers A1, A2, A4, A5, A6, A9, A10 and A11 are not affected by the routine, and available for the user. The program pointer is updated.

To obtain non standard parameters, i.e. a value, the following sequence of instructions is recommended:

```
LCR       AX, A12
ADKL      A12, 1
```

The routine I:RT1 is used to return control to the CREDIT module. The routines I:EVA0, I:EVA1, I:EVA3, I:EVA5, I:EVA7 and T:FDSP are called via the CF instruction, using A14 as stack pointer. The routine I.RT1, is called via the ABL instruction.

Registers A13, A14 and A15 must not be changed in an assembler subroutine.

Note: A number of standard assembler subroutines are held in the System library and may be called from CREDIT programs. They are described in appendix 4.

1.4.5 *Attach/Detach a device/file*

When a task wants to have exclusive access to a device or file, and locking out all other tasks from I/O at this device/file, the task has to execute an attach device instruction (DSC1, control code X'OE'). A time out value in multiples of 100 msec, must be specified for each attach to allow the monitor for supervising all attach requests and prevent (dead) lock situations. (DSC1, control code X'OB'). Time out value may be set to zero, then control is given immediately to the task which issued the request, with an indication in the extended status code whether the device of file is attached or not.

The attach function may be used to attach a data-file to a task. If index files are assigned to the data file, these index files must be attached too. When trying to attach a device, which is busy with I/O or already attached, the attach request is put in a device queue on first in first out principle within priority.

To release an attached device/file, the same task which issued the attach instruction, must perform a detach instruction (DSC1, control code X'OF').

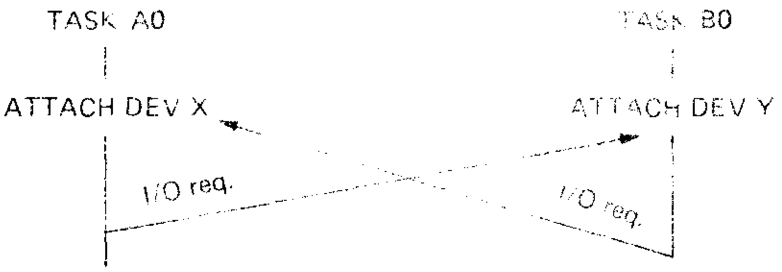
For an attached device, all I/O requests from other tasks for this device are queued in a device queue and will be executed after the device is detached.

All I/O on a device from a task which [illegible] instruction, is passing by the device queue ann will be [illegible]

When more than one device [illegible] attached to different tasks, the user program should be designed to prevent dead[illegible]

Such a situation will occur when [illegible] at the same time for each other to release attached resources.

Example: Device X is attached to task A0 and device Y is attached to task B0.
Task A0 issues an I/O request for device Y and B0 issues an I/O request for device X. Dead lock will be the result.



In this example dead lock can be avoided e.g. when task A0, before issuing the I/O request, for device Y, issues an attach request for device Y with a time-out value set. If device Y is not available, task A0 should temporarily detach device X before repeating the sequence again.

1.4.6 *Inter task communication*

This facility, if required, has to be included during system generation (SYSGEN). By means of the I/O instructions READ, WRITE, RREAD and RWRITE data can be transferred from one task to another task in the system. the appropriate inter task communication file codes must be assigned. The communication may be in addressed (RREAD, RWRITE) or unaddressed mode (READ, WRITE). The sending task is the one which issues the WRITE or RWRITE instruction and the receiving task is the one which issues the READ or RREAD instruction. No instructions are completed until there are two complementary instructions. (i.e. one READ and one WRITE.) This means that two complementary instructions must be issued by different tasks before any data transfer takes place and the instructions are completed. Different file codes must be assigned to input and output. This means that it is possible to configurate a task in three ways, as regards inter task communication:

1) Only input (READ, RREAD) possible – only input file code assigned.
2) Only output (WRITE, RWRITE) possible – only output file code assigned.
3) Both input and output possible – both input and output file codes assigned.

The task should only use the I/O file code assigned to it during system generation.

The file codes for input and/or output are declared in the DSET declarations as is done for I/O devices, one data set declaration for input and one for output.

The user is strongly recommended to assign the same file codes for inter task communication to all tasks, according to the single terminal interface principle.

When a task issues an inter task communication instruction, and no complementary instruction exists, the issued instruction is put into one of the four inter task communication queues, depending on whether the instruction was addressed to another task (RREAD, RWRITE) or unaddressed (READ, WRITE).

Two queues exist in the system, one for READ and one for WRITE (unaddressed). Only one of these queues may have one or more entries at any one time, since, as soon as they both contain an entry, the instructions are matched, communication takes place, and both instructions are completed.

In the case where a task issues a RREAD or RWRITE (addressed) to another task, and no complementary request exists, the issued instruction is queued on the addressed task. When the complementary instruction is issued, the instruction is completed and the request is removed from the queue.

The queueing principle for all inter task communication queues is on the FIFO (first in, first out) principle. This means that if a task issues e.g. a READ, it will be queued until any task issues a WRITE, or a RWRITE to this task, and then the matching is carried out and the instruction is completed. In case of a RREAD or RWRITE, naturally the first queued instruction may not be the matching one, i.e. it may be addressed to another task than the one which issued the current request. In this case the first request in the queue which is addressed to the current task is matched, and communication takes place.

If a READ, WRITE, RREAD or RWRITE instruction is to be supervised by the monitor in respect of time, a time out value should be set before the instruction is executed. Timing is set with the DSC1 instruction, with control code X'OB'. Different time out values in multiples of 100 msec, may be set for each instruction. These values are unique to the task which executed the time setting. The data-set-identifier, in the DSC1 instruction must refer to the corresponding DSET, for which the time must be set.

If no time out supervision is required, the binary data item in the DSC1 instruction, must be set to —1. If the value in this data item is set to zero, the request is completed immediately. No queueing is performed.

When the number of characters to be moved in two complementary instructions, is not equal, the smallest number of characters will be transferred. At completion of the instruction, the number of characters transferred will be returned.

1.4.7 *Notation*

The following symbols are used in the Instruction Reference Section (1.4.6).

PP	program pointer
=	equal to
≠	not equal to
>	greater than
⩾	greater than or equal to
≮	less than
⩽	less than or equal to
↔	compare
÷	divide (integer division)
x	multiply
+	add
-	subtract
(Operand)	the contents of operand
——	negate (the bar is written above the condition or value which is negated or complemented).

Examples:

(Operand-1) → operand-2	The contents operand-1 are stored in operand-2.
(Operand-1) ↔ (operand-2)	The contents of operand-1 are compared with the contents of operand-2.
(Operand-1) + (Operand-2)→ operand-1	The contents of operand-2 are added to the contents of operand-1 and the result is stored in operand-1.

1.4.8 *Instruction reference*

This section describes the syntax and use of each instruction. Intermediate object code is described for single data-items. For arrays one byte or two bytes must be added for each index referenced, depending on the addressing mode. When the ADRMOD option in the OPTNS directive equals two, data-item, data-set, literal constant, key table, picture and format references are extended with one byte in the intermediate object code. (For details about the object code format, when ADRMOD=1 or 2, see Appendix 8). The possible values of the variables in instructions are given in Appendix 1. The notation conventions are described in Section 1.1.5.

ABORT *Abort I/O request* ABORT

Syntax: [statement-identifier] ␣ ABORT ␣ data-set-identifier

Type: I/O instruction

Description: This function will abort a previously set I/O request (without wait) for a device indicated by data-set-identifier, in the same task. This request is only applicable to keyboard, typewriter, teller terminal printer, System Operator Panel (SOP) and intertask communications.

Condition register:
= 0 if abort is successful
= 2 if abort is not successful (e.g. I/O is already completed).

Condition mask:

0	1	2	3	4	5	6	7
SUCC	–	NOSUCC	–	$\overline{\text{SUCC}}$		$\overline{\text{NOSUCC}}$	Uncond-tional

Example: ABORT DSKBN

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	0	0	data set identifier					

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.
Operand-1 is a reference to a data set.
10/100 refers to the first data set.

ACTV *Activate* ACTV

Syntax: [statement-identifier] ⊔ ACTV ⊔ statement-identifier, task-identifier

Type: Scheduling instruction

Description: The task indicated by task-identifier is activated and execution is started at the instruction indicated by statement-identifier.
Task-identifier is a reference to a binary or string data item containing the task identity. In the case of a string data item, the two first bytes must contain the task identity.

Intermediate Code:

Byte 1	0 0 1 1	0 0 0 0
Byte 2	external reference	
operand-1	statement-identifier	
operand-2	task-identifier	

Byte 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.
operand-1 is a reference to the statement where execution has to be started.
operand-2 is a reference to a binary or string data item.

ADD *Add* ADD

Syntax: [statement-identifier] ADD data-item-identifier-1, $\left\{\begin{array}{l}\text{data-item-identifier-2}\\ \text{literal constant}\end{array}\right\}$

Type: Arithmetic instruction

Function: (Operand-1) + (Operand-2) → Operand-1

Description: Operand-2 is added to Operand-1 and the result is placed in Operand-1. Operand-2 is unchanged. Both operands must be binary or both operands must be decimal. A single data item may be used for both Operand-1 and Operand-2. In this case the data item is merely added to itself. The condition register is set according to the contents of Operand-1.

Condition Register:
= 0 if (Operand-1) = 0
= 1 if (Operand-1) > 0
= 2 if (Operand-1) < 0
= 3 if overflow

Condition mask:

0	1	2	3	4	5	6	7
=0	>0	<0	over-flow	≠0	≤0	≥0	uncon-ditional

Example:
ADD FIELD,=W'825' FIELD is declared as BIN
ADD WORK,=D'1' WORK is declared as BCD

Intermediate code format:

Byte 1	0	0	0	0	0	0	1	L
operand-1	data-item-identifier-1							
operand-2	data-item-identifier-2							

Byte 1 is operation code (X'02' or X'03')
L=0 operand-2 is a reference to a literal constant.
L=1 operand-2 is a reference to a binary or decimal data item.

ASSIGN

Assign a file

Syntax: [statement-identifier] ASSIGN data-set-identifier, control value, data-item-identifier, file-name-identifier, volume-name-identifier [, volume-name identifier] [, volume-name identifier] [, volume-name identifier]

Type: I/O instruction

Description: A file code as present in the data set referenced by data-set-identifier, is assigned to the file whose file-name is in the string-data-item referenced by file-name-identifier. The file name, in the data item, must be 8 bytes including trailing blanks. When control-value is 0, the file will be assigned as common file and is accessible by all tasks. When control value is 1 the file will be assigned as local file and is only accessible by the task issuing the assign. The data file may be extended over maximum four volumes. The volume name(s) are defined in the string data item(s) referenced by volume-name-identifier(s). Each data-item must contain 6 bytes for the volume name, including trailing blanks.

If an assignment is unsuccessful an error code is returned in the binary data item referenced by data-item identifier. The contents of this data item may be:

0	assignment successful performed
-1	Request error
1	Disk I/O error
2	No free entry in the device table
3	No file descriptor block available
4	One or more volumes unknown
5	File code already used
6	→ File name unknown
7	File section missing
8	Faulty disk format
9	more than 4 extents exist

Condition register:

=0	if assignment successful
=2	if assignment is unsuccessful

Condition mask:

0	1	2	3	4	5	6	7
SUCC	--	UNSUCC	-	$\overline{\text{SUCC}}$	--	$\overline{\text{UNSUCC}}$	Unconditional

Example: ASSIGN DFILE, 1, ERRCODE, FILEN, VOLNAM1, VOLNAM2

Intermediate code format:

<table>
<tr><td>Byte 1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Byte 2</td><td colspan="8">external reference</td></tr>
<tr><td>operand-1</td><td>0</td><td>0</td><td colspan="6">data-set identifier</td></tr>
<tr><td>operand-2</td><td colspan="8">control value</td></tr>
</table>

Continued

Intermediate code format: (continued)

operand-3	data-item-identifier
operand-4	file-name-identifier
Byte n	number of volumes
operand-5	volume-name-identifier
operand-6	volume-name-identifier
operand-7	volume-name-identifier
operand-8	volume-name-identifier

Bytes 1 and 2 are filled by the system.

Byte 2 contains a reference to an external system routine.

operand-1 is a reference to a data set.

10/100 refers to the first data set.

operand-2 is the control value (zero or one).

operand-3 is a reference to a binary data item.

operand-4 is a reference to a string data item.

Byte n contains a value filled by the translator.

operand 5, 6 are references to string data items.

ATTFMT *Attach Format* **ATTFMT**

Syntax: [statement-identifier] ␣ ATTFMT ␣ $\left\{\begin{array}{l}\text{format-list-identifier}\\ \text{data-item-identifier}\end{array}\right\}$

Type: Format control I/O

Description: The format list referenced by the format-list-identifier or the data-item-identifier, is attached to the current task. A previously attached format list will be detached. Only one format list may be current per task. Data-item-identifier refers to a string data-item, which item contains characters forming together a valid format list. This instruction is only used, when the format list contains input fields which are supposed to receive data from a keyboard.

Condition register: Unchanged

Example: ␣ ATTFMT ␣ FRMT1

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	L
Byte 2	external reference							
operand-1	format-list-identifier							

Bytes 1 and 2 are filled by the system.

operand-1 is a reference to a format list (L=1) or to a string data-item (L=0).

B *Branch* **B**

Syntax: [statement-identifier] ⊔ B ⊔ $\left[\begin{Bmatrix}\text{equate-identifer,}\\ \text{condition mask,}\end{Bmatrix}\right]$ statement-identifier

Type: Branch instruction

Description: The instruction to be executed is indicated by statement-identifier, if operand-1 matches the contents of the condition register. Else, the instruction following the branch will be executed. If operand-1 is omitted an unconditional branch (value 7) is generated.
The translator decides whether a shortbranch or longbranch should be generated, depending on the branch target.

Condition register: not changed.

Example: B ⊔ INP3
B ⊔ 2,INP4

Intermediate code format:
(long branch)

Byte 1	0	0	1	1	1	CND
Byte 2	index to T:BAT					

Byte 1 is the operation code (X'38' up to X'3F')
CND is the condition mask field
Byte 2 contains an index to a branch address table (T:BAT)

Intermediate code format:
(short branch)

Byte 1	0	1	0	1	B	CND
Byte 2	displacement					

Byte 1 is the operation code (X'50' up to X'5F')
B = 0 forward branching
B = 1 backward branching
CND is the condition mask field
Byte 2 contains the displacement

BBEOD *Branch on begin/end device* BBEOD

Syntax: [statement-identifier] ⊔ BBEOD ⊔ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is three (begin or end of device), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed.
This instruction should be used after an I/O instruction.
The translator decides whether a shortbranch or longbranch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BBEOD⊔DEVERR

Intermediate
code format:
(long branch)

Byte 1	0	0	1	1	1	0	1	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3B')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate
code format:
(short branch)

Byte 1	0	1	0	1	B	0	1	1
Byte 2	displacement							

Byte 1 is the operation code (X,5B', X'53')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BE *Branch on equal* BE

Syntax: [statement-identifier] ␣ BE ␣ statement-identifier

Type: Branch instruction.

Description: If the contents of the condition register is zero (equal), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed. This instruction should be used after a comparison. The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BE␣EQUAL

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	0	0
Byte 2	index to T:BAT							

Byte 1 is the operation code X'38')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	0	0
Byte 2	displacement							

Byte 1 is the operation code (X'58', X'50')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BEOF *Branch on End of File* **BEOF**

Syntax: [statement-identifier] ␣ BEOF ␣ statement-identifier.

Type: Branch instruction

Description: If the contents of the condition register is one (end of File), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an I/O instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BEOF ENDOFF

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	0	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'39')
Byte 2 contains an index to a branch address table (T:BAT)

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	0	1
Byte 2	displacement							

Byte 1 is the operation code (X'59', X'51')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement

BERR *Branch on Error* BERR

Syntax: [statement-identifier] ␣ BERR ␣ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is two (Error), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed. This instruction should be used after an I/O instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BERR␣ERROR1

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	1	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3A')
Byte 2 contains an index to a branch address table (T:BAT)

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	1	0
Byte 2	displacement							

Byte 1 is the operation code (X'5A', X'52')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BG *Branch on greater* BG

Syntax: [statement-identifier] ⊔ BG ⊔ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is one (greater), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed. This instruction should be used after a comparison. The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BG⊔GREATER

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	0	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'39')
Byte 2 contains an index to a branch address table (T:BAT)

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	0	1
Byte 2	displacement							

Byte 1 is the operation code (X'59', X'51')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BL *Branch on less* BL

Syntax: [statement-identifier] ␣ BL␣statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is two (less), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed.
This instruction should be used after a comparision.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BL␣LESS

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	1	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3A')
Byte 2 contains an index to a branch address table (T:BAT)

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	1	0
Byte 2	displacement							

Byte 1 is the operation code (X'5A', X'52')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BN *Branch on negative* BN

Syntax: [statement-identifier] ␣ BN ␣ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is two (negative), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed.
This instruction should be used after an arithmetic instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BN ␣ NEG

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	1	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3A')
Byte 2 contains an index to a branch address table T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	1	0
Byte 2	displacement							

Byte 1 is the operation code (X'5A', X'52')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement

BNE *Branch on not equal* BNE

Syntax: [statement-identifier] ␣ BNE ␣ statement-identifier.

Type: Branch instruction

Description: If the contents of the condition register is unequal to zero (not equal), the program will branch to the instruction indicated by statement-identifier.
This instruction should be used after a comparision.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNE ␣ UNEQ

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	1	0	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3C')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	1	0	0
Byte 2	displacement							

Byte 1 is the operation code (X'5C', X'54')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BNEOF *Branch on no End of file* BNEOF

Syntax: [statement-identifier] ⊔ BNEOF ⊔ statement-identifier

Type: Branch instruction.

Description: If the contents of the condition register is unequal to one (Not End of file), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an I/O instruction.
The translator decides whether a short branch or long branch should be generated depending on the branch target.

Condition register: Unchanged

Example: BNEOF ⊔ NOTEOF

Intermediate code format:
(long branch)

Byte 1	0	0	1	1	1	1	0	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3D')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format:
(short branch)

Byte 1	0	1	0	1	B	1	0	1
Byte 2	displacement							

Byte 1 is the operation code (X'5D', X'55')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BNERR   *Branch on no error*   BNERR

Syntax: [statement-identifier] ␣ BNERR ␣ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is unequal to two (No Error), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an I/O instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNERR ␣ NOERR

Intermediate code format:
(long branch)

Byte 1	0	0	1	1	1	1	1	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3E')
Byte 2 contains an index to a branch address table (T:BAT)

Intermediate code format:
(short branch)

Byte 1	0	1	0	1	B	1	1	0
Byte 2	displacement							

Byte 1 is the operation code (X'5E', X'56')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BNG *Branch on not greater* BNG

Syntax: [statement-identifier] ␣ BNG ␣ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is unequal to one (not greater), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after a comparision.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNG ␣ NOTGRT

Intermediate
code format:
(long branch)

Byte 1	0	0	1	1	1	1	0	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3D')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate
code format:
(short branch)

Byte 1	0	1	0	1	B	1	0	1
Byte 2	displacement							

Byte 1 is the operation code (X'5D', X'55')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement

BNL *Branch on not less* BNL

Syntax: [statement-identifier] ⊔ BNL ⊔ statement-identifier.

Type: Branch instruction

Description: If the contents of the condition register is unequal to two (not less), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after a comparision.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNL ⊔ NOTLESS

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	1	1	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3E')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	1	1	0
Byte 2	displacement							

Byte 1 is the operation code (X'5E', X'56')
 B = 0 forward branching
 B = 1 backward branching
Byte 2 contains a displacement.

BNN — *Branch on not negative* — BNN

Syntax: [statement-identifier] ⊔ BNN ⊔ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is unequal to two (not negative), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an arithmetic instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNN ⊔ NOTNEG

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	1	1	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3E')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	1	1	0
Byte 2	displacement							

Byte 1 is the operation code (X'5E', X'56')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BNOK *Branch on not OK* BNOK

Syntax: [statement-identifier] ⊔ BNOK ⊔ statement-identifier.

Type: Branch instruction

Description: If the contents of the condition register is unequal to zero (not oké), the program with branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an I/O instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNOK ⊔ NOTOKE

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	1	0	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3C')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	1	0	0
Byte 2	displacement							

Byte 1 is the operation code (X'5C, X'54')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement

BNP — *Branch on not positive* — BNP

Syntax: [statement-identifier] ⊔ BNP ⊔ statement-identifier.

Type: Branch instruction

Description: If the contents of the condition register is unequal to one (not positive), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an arithmetic instruction.
The translator decides whether a short branch or long branch should generated, depending on the branch target.

Condition register: Unchanged.

Example: BNP ⊔ NOTPOS

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	1	0	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3D')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	1	0	1
Byte 2	displacement							

Byte 1 is the operation code (X'5D', X'55')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement

BNZ *Branch on not zero* BNZ

Syntax: [statement-identifier] ⊔ BNZ ⊔ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is unequal to zero (not zero), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an arithmetic instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BNZ⊔NONZER

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	1	0	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3C')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	1	0	0
Byte 2	displacement							

Byte 1 is the operation code (X'5C'), X'54')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BOFL *Branch on overflow* BOFL

Syntax: [statement identifier] ␣ BOFL ␣ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is three (overflow), the program will branch to the instruction indicated by statement-identifier.
Otherwise the instruction following the branch will be executed.
This instruction should be used after an arithmetic instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged

Example: BOFL ␣ OVERFL

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	1	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'3B')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	1	1
Byte 2	displacement							

Byte 1 is the operation code (X'5B', X'53')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BOK Branch on OK BOK

Syntax: [statement-identifier] ␣ BOK ␣ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is zero (oke), the program will branch to the instruction indicated by statement identifier. Otherwise the instruction following the branch will be executed.
This instruction should be used after an I/O instruction.
The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BOK ␣ OKE

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	0	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'38')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	0	0
Byte 2	displacement							

Byte 1 is the operation code (X'58', X'50')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement

BP *Branch on positive* **BP**

Syntax: [statement-identifier] ⊔ BP ⊔ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is one (positive), the program will branch to the instruction indicated by statement-identifier. Otherwise the instruction following the branch will be executed. This instruction should be used after an arithmetic instruction. The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BP ⊔ POS

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	0	1
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'39')
Byte 2 contains an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	0	1
Byte 2	displacement							

Byte 1 is the operation code (X'59', X'51')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

BZ *Branch on zero* BZ

Syntax: [statement-identifier] ⊔ BZ ⊔ statement-identifier.

Type: Branch instruction.

Description: If the contents of the condition register is zero (zero), the program will branch to the instruction indicated by statement identifier. Otherwise the instruction following the branch will be executed. This instruction should be used after an arithmetic instruction. The translator decides whether a short branch or long branch should be generated, depending on the branch target.

Condition register: Unchanged.

Example: BZ⊔ZERO

Intermediate code format: (long branch)

Byte 1	0	0	1	1	1	0	0	0
Byte 2	index to T:BAT							

Byte 1 is the operation code (X'38')
Byte 2 contains an index to branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	0	0	0
Byte 2	displacement							

Byte 1 is the operation code (X'58', X'50')
B = 0 forward branching
B = 1 backward branching
Byte 2 contains a displacement.

CALL *Call* **CALL**

Syntax: [statement-identifier]␣CALL␣subroutine-identifier [,actual-parameter-list] . . .

Type: Subroutine control instruction

Description: Control is given to a subroutine written in assembly language. Thus, subroutine-identifier must be declared as an external identifier (EXT).

Actual parameters which can be passed to the subroutine, in addition to the parameters listed in the CREDIT syntax definition, include data set identifiers. There is no limit on the number of parameters which can be passed. Parameters may also be passed, using the CON directive. Literal constant parameters of the type 'X' are not allowed.

The programmer is responsible for a correct return from the assembly routine to the interpreter. Hence when parameters are passed to the subroutine, the program pointer has to be updated by the assembly routine, prior to control being given back to the interpreter.

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	L
operand-1	subroutine-identifier							
operand-2	parameter							

Byte 1 is the operation code (X'30').
operand-1 is an external reference to the subroutine.
operand-2 etc. are available for passing parameters.

CBE — *Compare and branch on equal* — CBE

Syntax: [statement-identifier] ␣CBE␣ data-item-identifier-1 $\left\{\begin{array}{l}\text{data-item-identifier-2}\\\text{literal constant}\end{array}\right\}$,statement-identifier

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').

b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

If the contents of both operands are equal, the next instruction to be executed is found at the address specified by statement-identifie
If the contents of both operands are not equal, the instruction following the compare and branch equal (CBE) will be executed. Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch on equal.
Operand-1 and operand-2 must refer to the same type of data item decimal, binary or string.

Example:

```
CBE    INLEN, $MIN,RDERR2
CBE    INLEN, CBINO, RDERR2
```

Condition register: If both identifiers are references to numeric data items.

= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRINGI

= 0 if (Operand-1) = (Operand-2)

Intermediate code format:

Byte 1		1 B	B	0	0	0	L
operand-1	data-item-identifier-1						
operand-2	data-item-identifier-2						
Byte n	displacement						

Byte 1 is the operation code (X'10', X'11', X'20', X'21')
B = 0 forward branching
B = 1 backward branching
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to a data item.
operand-1 and operand-2 are references to data items of the type decimal, binary or string.
Byte n contains a displacement.

CBG *Compare and branch on greater* CBG

Syntax: [statement-identifier] ␣CBG␣ data-item-identifier-1, $\left\{\begin{array}{l}\text{data-item-identifier-2}\\ \text{literal constant}\end{array}\right\}$,statement-identifier

Type: Branch instruction.

Function: (Operand-1) : (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').

b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

If the contents of operand-1 is not greater than the contents of operand-2, the instruction following the compare and branch on greater (CBG) will be executed.

Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch on greater

Operand-1 and operand-2 must refer to the same type of data item decimal, binary or string.

Example:

```
CBG    INLEN, $MIN, RDERR3
CBG     INLEN, CBINO, RDERR3
```

Condition register. If both identifiers are references to numeric data items.

= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRINGI

= 0 if (Operand-1) = (Operand-2)

Continued

Intermediate
code format:

Byte 1	0 0	1-B	B	0 0 1	L
operand-1	data-item-identifier-1				
operand-2	data-item-identifier-2				
Byte n	displacement				

Byte 1 is the operation code (X'12', X'13', X'22', X'23')
B = 0 forward branching
B = 1 backward branching
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to a data item.
operand-1 and operand-2 are references to data-items of the type decimal, binary or string.
Byte n contains a displacement.

CBL *Compare and branch on less* CBL

Syntax: [statement identifier] ␣ CBL ␣ data-item-identifier-1, $\left\{\begin{array}{l}\text{data-item-identifier-2}\\ \text{literal constant}\end{array}\right\}$,statement-identifier

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

If the contents of operand-1 is less than the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
If the contents of operand-1 is not less than the contents of operand-2, the instruction following the compare and branch on less (CBL) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch on less (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch.
Operand-1 and operand-2 must refer to the same type of data item – decimal, binary or string.

Example:

```
CBL    INLEN, $MIN, RDERR4
CBL     INLEN, CBINO, RDERR4
```

Condition register: If both identifiers are references to numeric data items.

= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRING!

= 0 if (Operand-1) = (Operand-2)

Intermediate code format:

Byte 1	0	0	1-B	B	0	1	0	L
operand-1	data-item-identifier-1							
operand-2	data-item-identifier-2							
Byte n	displacement							

Byte 1 is the operation code (X'14', X'15', X'24', X'25')
B = 0 forward branching
B = 1 backward branching
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to a data item.
operand-1 and operand-2 are references to data-items of the type decimal, binary or string
Byte n contains a displacement.

CBNE   *Compare and branch on not equal*   CBNE

Syntax: [statement-identifier] ␣ CBNE ␣ data-item-identifier-1, {data-item-identifier-2 / literal constant}

,statement-identifier

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2.
The condition register is set according to the result of this comparison.
When the two data items have a different size, the comparison will be executed as follows:

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').

b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

If the contents of operand-1 is unequal to the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
In all other cases the instruction following the compare and branch on not equal (CBNE) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch on not equal (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch.
Operand-1 and operand-2 must refer to the same type of data item – decimal, binary or string.

Example:

```
CBNE  INLEN, $MIN, RDERR5
CBNE  INLEN, CBINO, RDERR5
```

Condition register: If both identifiers are references to numeric data items.

= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operand's are of the type STRING or STRINGI

= 0 if (Operand-1) = (Operand-2)

Intermediate
code format:

Byte 1	0	0	1-B	B	1	0	0	L
operand-1	data-item-identifier-1							
operand-2	data-item-identifier-2							
Byte n	displacement							

Byte 1 is the operation code (X'18', X'19', X'19', X'28', X'29')
B = 0 forward branching
B = 1 backward branching
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to a data item
operand-1 and operand 2 are references to data-items of the type decimal, binary or string.
Byte n contains a displacement.

CBNG   *Compare and branch not greater*   CBNG

Syntax: [statement-identifier] ␣CBNG␣ data-item-identifier-1,{data-item-identifier-2 / literal constant} ,statement-identifier

Type: Branch instruction.

Function: (Operand-1) :: (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. *The condition register is set according to the result of this comparison.* When the two data items have a different size, the comparison will be executed as follows:

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').

b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

If the contents of operand-1 is not greater than the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
In all other cases, the instruction following the compare and branch on not greater (CBNG) will be executed.
Statement identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch on not greater (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch.
Operand-1 and operand-2 must refer to the same type of data item – decimal, binary or string.

Example:

```
CBNG     INLEN, $MIN, RDERR6
CBNG     INLEN, CBINO, RDERR6
```

Condition register: If both identifiers are references to numeric data items.

= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) >(Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRINGI

= 0 if (Operand-1) = (Operand-2)

Intermediate code format:

<table>
<tr><td>Byte 1</td><td>0</td><td>0</td><td>1-B</td><td>B</td><td>1</td><td>0</td><td>1</td><td>L</td></tr>
<tr><td>operand-1</td><td colspan="8">data-item-identifier-1</td></tr>
<tr><td>operand-2</td><td colspan="8">data-item-identifier-2</td></tr>
<tr><td>Byte n</td><td colspan="8">displacement</td></tr>
</table>

Byte 1 is the operation code (X'1A', X'1B', X'2A', X'2B')

B = 0 forward branching
B = 1 backward branching
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to a data item.
operand-1 and operand-2 are references to data-items of the type decimal, binary or string.
Byte n contains a displacement.

CBNL *Compare and branch on not less* CBNL

Syntax: [statement-identifier] ⌴CBNL⌴ data-item-identifier-1, {data-item-identifier-2 / literal constant}

,statement-identifier

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows.

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').

b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

If the contents of operand-1 is not less than the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.

If the contents of operand-1 is not lesses than the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
In all other cases, the instruction following the compare and branch on less (CBNL) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch on not less (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch.
Operand-1 and operand-2 must refer to the same type of data item – decimal, binary or string.

Example:

```
CBNL      INLEN, $MIN, RDERR7
CBNL      INLEN, CBINO, RDERR7
```

Condition register: If both identifiers are references to numeric data items.

= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRINGI

= 0 if (Operand-1) = (Operand-2)

Continued

Intermediate code format:

Byte 1	0	0	1-B	B	1	1	0	L
operand-1	data-item-identifier-1							
operand-2	data-item-identifier-2							
Byte n	displacement							

Byte 1 is the operation code (X'1C', X'1D', X'2C', X'2D').

B = 0 forward branching
B = 1 backward branching

L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to a data item.
operand-1 and operand-2 are references to data-items of the type decimal, binary or string.
Byte n contains a displacement.

CLEAR *Clear* CLEAR

Syntax: [statement-identifier] ⊔CLEAR⊔ data-item-identifier

Type: Logical instruction.

Function: 0 → data-item-identifier

Description: The content of data-item-identifier is set to zero (FALSE).

Data-item-identifier must refer to a boolean data item. (Length 1 bit)

The condition register is set according to the previous value of data-item-identifier.

Condition register: = 0 if (data-item-identifier) = 0

Condition mask:

0	1	2	3	4	5	6	7
=0	–	–	–	≠0	–	–	–

Intermediate code format:

Byte 1	0	1	0	0	0	0	0	0
Operand-1	data-item-identifier							

Byte 1 is the operation code (X'40').
Operand-1 is a reference to a boolean data item.

CMP Compare CMP

Syntax: [statement-identifier] ␣CMP␣ data-item-identifier-1, {data-item-identifier-2 | literal constant}

Type: Arithmetic instruction

Function: (Operand-1) − (Operand-2)

Description: Operand-1 is compared with Operand-2.
The condition register is set according to the result of this comparison.
When the two data items have a different size, the comparison will be executed as follows:

a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

Both operands must be the same type of data item — decimal, binary or string.

Condition register: If both operands are numeric or string data items:

= 0 if (operand-1) = (operand-2)
= 1 if (operand-1) > (operand 2)
= 2 if (operand-1) < (operand-2)

Condition mask:

0	1	2	3	4	5	6	7
Op1=Op2	Op1>Op2	Op1<Op2	–	Op1≠Op2	Op1≤Op2	Op1≥Op2	uncon-ditional

Example:

```
CMP   FIELD1, FIELD2,   FIELD1 and FIELD2 are declared as BIN.
CMP   BANKID, NAME      BANKID and NAME are declared as STRG.
CMP   BANKID, = C'BANK'
```

Intermediate code format:

Byte 1	0	0	0	0	0	1	1	L
operand-1	data-item-identifier-1							
operand-2	data-item-identifier-2							

Byte 1 is the operation code (X'06' or X'07')
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to data-item-identifier-2, array-identifier-2 or a formal parameter.

COPY Copy COPY

Syntax: [statement-identifier] ␣COPY ␣ data-item-identifier-1, pointer-identifier-1,size-identifier,data-item-identifier-2,pointer-identifier-2

Type: String instruction

Function: (Operand-4) → Operand-1
(Pointer- (pointer-
identifier 2) identifier-1)

Description: Starting at pointer identifier-2, the content of operand-4 is copied from left to right to operand-1 beginning at pointer-identifier-1.
The number of decimal digits or bytes to be copied is specified by size-identifier.
This COPY is only possible between two decimal data items or two string data items. Between two decimal data-items is copied on digit base. In the other case it is copying on byte base.
The first characters of operand-1 and operand-4 are counted as zero when setting the pointer.

Condition register: Not significant.

Example: COPY FIELD1, P1, LNGTH, FIELD2, P2

Intermediate code format:

Byte 1	0	1	1	0	0	0	1	0
operand-1	data-item-identifier-1							
operand-2	pointer-identifier-1							
operand-3	size-identifier							
operand-4	data-item-identifier-2							
operand-5	pointer-identifier-2							

Byte 1 is the operation code (X'62')
operand-1 and operand-4 are references to string al data items.
data items or decimal data items
operands-2,3,5 are references to binary data items.

Delay

Syntax: [statement-identifier] ⊔ DELAY ⊔ data-item-identifier

Type: Scheduling instruction

Description: Execution of the running task is delayed. The delay time is specified in multiples of 100 msec in a binary data item indicated by data-item-identifier.

Condition register: Unchanged.

Example: DELAY ⊔ DELTIM

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	data item identifier							

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.
operand-1 is a reference to a binary data item.

DETFMT *Detach format* DETFMT

Syntax: [statement-itenditier] ␣ DETFMT ␣

Type: Format control I/O.

Description: The format attached to the current task is detached.

Condition Register: Unchanged

Intermediate Code Format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							

Bytes 1 and 2 are filled by the system.

Syntax: [statement-identifier] DISPLAY control-value,

data-item-identifier-1 { data-item-identifier-2 / literal constant }

Type: Format control

Description: Depending on control value one of the following operations on the current format list is performed.

control value — Significance

0 The format lines starting from the line number contained in the binary data item, referenced by data-item-identifier-1, to the line number contained in the binary data item referenced by data-item-identifier 2 are displayed. When the second line number (referenced by data-item-identifier-2) is zero then all lines of the current format list are displayed starting at the line number contained in the data-item referenced by data-item-identifier-1.
Both data-items may contain the same line number,

1 The FKI-input fields of the current format list are displayed on the screen in the appropriate positions, using the FKI-input field numbering sequence. Data-item identifier-1 refers to a binary data-item containing the FKI-input field number, from which displaying starts. Data-item identifier 2 refers to a binary data item containing the FKI-input field number at which displaying stops. When this data item contains zero all FKI-input fields will be displayed, starting at FKI-input field number contained in the data item referenced by data-item-identifier 1. The prompts are not erased.
Both data-items may contain the same line-number.

2 Similar to control value 1, but the FINP input fields are now displayed, using the FINP input field numbering. The prompts are not erased.

3 Similar to control value 1, but both FKI-input fields and FINP-input fields are displayed using the general field numbering.
The prompts are not erased.

4 Similar to control value 1, but screen is not cleared. The last line number to be displayed may also be indicated by a literal constant of the type binary.

Condition register: = 0 if OK
= 2 if ERROR

Condition mask:

0	1	2	3	4	5	6	7
OK	–	ERROR	–	$\overline{\text{OK}}$	–	$\overline{\text{ERROR}}$	Unconditional

Example: DISPLAY ␣ 0, LINE 3, LINE 12

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	L
Byte 2	external reference							
operand-1	control value							
operand-2	data-item-identifier-1							
operand-3	data-item-identifier-2							

Bytes 1 and 2 are filled by the system.
operand-1 is the control value.
operand-2 and operand-3 are references to binary data items.
L=1 operand-3 is a reference to a literal constant.
L=0 operand-3 is a reference to a data item.

DIV *Divide* DIV

Syntax: [statement-identifier] ␣ DIV ␣ data-item-identifier-1, $\left\{\begin{array}{l}\text{data-item-identifier-2}\\ \text{literal constant}\end{array}\right\}$

Type: Arithmetic instruction

Function: (Operand-1) ÷ (Operand-2) → Operand-1

Description: Operand-1 is divided by operand-2 and the result is stored in operand-1.
Operand-2 is unchanged. Both operands must be decimal or binary. The remainder is lost. Division by zero results in overflow and operand-1 is set to zero.

Condition register: = 0 if (operand-1) = 0
= 1 if (operand-1) > 0
= 2 if (operand-1) < 0
= 3 if overflow

Example:

```
DIV    WORK,=D'+4'
DIV    FIELD, FIELD2
```

Intermediate code format:

Byte 1	0	0	0	0	1	0	1	L
operand-1	data-item-identifier-1							
operand-2	data-item-identifier-2							

Byte 1 is the operation code (X'0A' or X'0B').
L=1 operand-2 is a reference to a literal constant
L=0 operand-2 is a reference to data-item

DLETE *Delete* DLETE

Syntax: [statement-identifier] ␣ DLETE ␣ data-item-identifier,pointer-identifier,size-identifier

Type: String instruction

Function: delete (operand-1)
(pointer-identifier)

Description: Starting at pointer-identifier, the contents of operand-1 are deleted from left to right.
The number of characters to be deleted is specified by size-identifier.

The remaining characters at the right of the deletion are shifted left. The number of shift positions corresponds with the content of size-identifier. Space characters are inserted from the right.

Operand-1 must be a string data item. The first character of operand-1 is counted as zero when setting the pointer.

Condition register: Not significant.

Example: DLETE DIELD,P1,L1

Intermediate code format:

Byte 1	0	1	1	0	0	1	1	0
operand-1	data-item-identifier-1							
operand-2	pointer-identifier							
operand-3	size-identifier							

Byte 1 is the operation code (X'66').
Operand-1 refers to a string data item.
Operands-2,3 refer to binary data items.

DSC1 *Dataset control one* **DSC1**

Syntax: [statement-identifier] ␣ DSC1 ␣ [,NW,] data-set-identifier, $\left\{\begin{array}{l}\text{control value}\\ \text{equate-identifier}\end{array}\right\}$

,data-item-identifier

Type: I/O instruction

Description: This statement is used to control a data set indicated by data-set-identifier.

The kind of control is specified by operand-3, the values of which are found in the control code table 1 (see below).

The binary or decimal data item indicated by operand-4 contains device dependent control information. ,NW, indicates that the no wait option is required.

Condition register:

= 0 if I/O successful	(OK)
= 1 if End of file	(EOF)
= 2 if Error	(ERR)
= 3 if Begin or end of device	(BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{\text{OK}}$	$\overline{\text{EOF}}$	$\overline{\text{ERR}}$	uncon-ditional

Example: DSC1 DSSOPO,OFF,ALLAMP

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	data-set-identifier						
operand-2	control value							
operand-3	data-item-identifier							

Bytes 1 and 2 are filled by the system.
Byte 2 contains a reference to an external system routine.
W is the wait bit.
W=0 no wait
W=1 wait
Operand-1 is a reference to the relevant data set.
10/100 refers to the first data set.
Operand-2 is a hexadecimal integer, which corresponds with the control code.
Operand-3 is a reference to a binary or decimal data item.

Continued

Control Code	Data set Device	Significance	Recommended value identifier
00	TK,MT DI,SOP,KI DC TV	Load cassette/tape Turn on indicator Transfer parameters Transf. doc. param. (PTS6371)	LOAD ON TRPAR
01	DI,SOP,KI DC	Turn off indicator Set status	OFF SSTAT
02	DY	Erase display line	ERASE
06	TV,DY	Position voucher/ pass book. (Number of line steps) Position cursor PTS6371, line number	POS LINNO
08	DL [1]	Random delete	DEL
0B	II,IO,DC SOP,DI,KI	Set time out value Set indicator flashing	STIMO FLASH
0C	DL [1]	Get currency (data)	GCD
0D	DL [1] TV,TJ	Get currency (index) Set printer parameters	GCD SETPAR
0E		Attach device/file (wait bit must be set)	ATTACH
0F		Detach device/file (wait bit must be set)	DETACH
10	DL	Delete record and index	IDEL
11-17,19-FF		Reserved for future use	

[1] DL = logical disk file

Device dependent control information.

Control code	Device type	Data item contains	Significance
00	TK	1	Cassette without sequence number on tape.
		0	Cassette with sequence number on tape.
00 01	SOP		Light positions corresponding with the one bits in the binary data item are turned on/off. Other lights are not altered. The right most panel light corresponds with bit 15 in the binary data item.
00 01	DI/KI	PTS 6241 and 6242.	Light positions corresponding with the one bits in the binary data item are turned on/off. Other lights are not altered. Lamp L1 on each device. B=1, Bell is sounded at the keyboard.

PTS 6241 and 6242.

0		L1	L2	L3	L4	L5	L6	L7	L8
0	7	8	9	10	11	12	13	14	15

PTS 6232 and 6234

0						L4	L3	L2	L1
0	7	8	9	10	11	12	13	14	15

PTS 6233

B		L8	L7	L6	L5	L4	L3	L2	L1
0	7	8	9	10	11	12	13	14	15

PTS 6331:

0							L3	L2	L1
0	7	8	9	10	11	12	13	14	15

PTS6236, PTS6271, PTS6272

0		L1	L2	L3	L4	L5	L6
0		10	11	12	13	14	15

<table>
<tr><th>Control code</th><th>Device type</th><th>Data-item contains:</th><th>Significance</th></tr>
<tr><td>00</td><td>DC</td><td><table><tr><td>0</td><td>7</td><td>8</td><td>15</td></tr><tr><td>0</td><td>0</td><td colspan="2">TASK Address</td></tr></table>
<table><tr><td>0</td><td>7</td><td>8</td><td>15</td></tr><tr><td colspan="2">TC-Select address</td><td colspan="2">TC-poll address</td></tr></table></td><td>When issued from normal task

When issued from a DC-task</td></tr>
<tr><td>00</td><td colspan="3">Badge card reader lamps
<table><tr><td>0</td><td></td><td>L2</td><td>L1</td></tr><tr><td>0</td><td></td><td>14</td><td>15</td></tr></table>
<table><tr><th>L2</th><th>L1</th><th>effect</th></tr><tr><td>0</td><td>0</td><td>no action</td></tr><tr><td>0</td><td>1</td><td>lamp is turned on (input from BCR)</td></tr><tr><td>1</td><td>0</td><td>not valid</td></tr><tr><td>1</td><td>1</td><td>flash lamps (input from PIN keyboard)</td></tr></table></td></tr>
<tr><td>01</td><td colspan="3"><table><tr><th>L2</th><th>L1</th><th>effect</th></tr><tr><td>0</td><td>0</td><td>no action</td></tr><tr><td>0</td><td>1</td><td>if lamp on, turn lamp off
if lamp flashing, not valid
if lamp off, no action</td></tr><tr><td>1</td><td>0</td><td>if lamp on, no action
if lamp flashing, turn lamp on
if lamp off, no action</td></tr><tr><td>1</td><td>1</td><td>if lamp on, turn lamp off
if lamp flashing, turn lamp off
if lamp off, no action</td></tr></table></td></tr>
<tr><td>01</td><td>DC</td><td></td><td>Set status</td></tr>
<tr><td>02</td><td>DY</td><td>Number</td><td>A number of characters as specified in the binary data-item, are erased from the current cursor position. Only characters on the same line where the cursor is positioned can be erased. The cursor remains in its original position. The maximum number of characters that can be erased is as follows:
PTS6344 –80 [illegible]
PTS6351 –36
PTS6385 –40
PTS6386 –40</td></tr>
</table>

Device dependent control information

Control code	Device type	Data-item contains	Significance
06	DY	LINE NUMBER | COLUMN 0 7 8 15	Cursor is sent to line and column numbers as specified in the binary data item. X'0000' is the cursor home position. PTS 6344 – [illegible] lines, 64 char per line (6342 – 24 lines, 80 char per line) PTS 6351 (alphanumeric version) – 8 lines, 36 char per line. PTS6355 1 line of 40 characters PTS6366 9 lines of 40 characters No information is erased
06	TV	number of line steps.	Position voucher/passbook by giving the number of line feed steps (2 steps one line), in the binary data-item.
08	DL	Logical record number.	The status character is set to "FREE" on the record, the logical record number of which is in the binary data item. Delete is only allowed on a record which is under exclusive access Exclusive access is released after function (No check is performed to detect if the record was already "FREE")
0B	II, IO	Number	Set time in multiples of 100 msec for intertask communication, attach/detach device/file or data communication.
0B	SOP,DI KI	PTS 6241 and 6242	Light positions corresponding with the one bits in the binary data item are lit once every second.

0	L1	L2	L3	L4	L5	L6	L7	L8
7	8	9	10	11	12	13	14	15

Device dependent control information:

Control code	Device type	Data-item contains:	Significance
0B	SOP DI KI	PTS 6232 and 6234 (see below) PTS 6233 (see below) PTS 6331 (see below) PTS 6236 (see below)	Lamp L1 is the left most lamp on each device.

PTS 6232 and 6234

0						L4	L3	L2	L1
	7	8	9	10	11	12	13	14	15

PTS 6233

B		L8	L7	L6	L5	L4	L3	L2	L1
	7	8	9	10	11	12	13	14	15

If B=1, a buzzer is sounded at the keyboard.

PTS 6331:

0							L3	L2	L1
	7	8	9	10	11	12	13	14	15

PTS 6236:

L1	L2	L3	L4	L5	L6
10	11	12	13	14	15

Control code	Device type	Data-item contains:	Significance
0C	DL	Current Record Number (Data record)	Current Record Number of a data file is returned in the binary data item.
0D	DL	Current Record Number (Index record)	Current Record Number of a index file is returned in the binary data item
0D	TV,TJ (PTS6371)		

With this instruction it is possible to change one or more of the following parameters:

- Upper/Upper and Lower case character set (L)
- National character variation (NCV)
- Character pitch document/journal (CPD/CPJ)

The first two parameters are the same for both the journal and the document station, but the character pitch may have different values for the two stations. All the parameters may be set up in one request issued to only one of the devices. This instruction is only intended for use where the parameters have to be changed during the running of the application; if they are fixed, they should be specified during system generation.

<table>
<tr><th>Control code</th><th>Device type</th><th>Data-item contains:</th><th>Significance</th></tr>
<tr><td colspan="4">The data items contains the parameter information, as follows:
<table>
<tr><td>0 3</td><td>4 7</td><td>8 11</td><td>12 15</td></tr>
<tr><td>L</td><td>NCV</td><td>CPJ</td><td>CPD</td></tr>
</table>
where : L is the case indicator;
if zero, no change is required;
if set to eight only upper case characters are required. Any code in the range /60—/7E is printed as the corresponding capital letter. If set to nine, both capitals and lower case are printed; the height of the capitals is reduced from 2.7 mm to 2.1 mm by using seven dots instead of nine.
NCV is set within the range 0—A for the national character variations, as shown in the tablet at the end of this DSC1 description.
CPJ and CPD are the character pitch for the journal and document stations respectively; if set to zero, no change is required.
The pitch may be 4, 5 or 6, corresponding with 15 char/inch, 12 char/inch, or 10 char/inch respectively.
If any of the parameters have an illegal value, none of the parameters will be set, and the request is completed with CR = 2 (Error).
If the printer is not operable for any reason, the request is completed with CR = 2 (Error); in this case the parameters are stored and sent to the printer when power is restored, but in practise they should be sent again, unless an XSTAT shows that this was the only cause of the CR being set to 2.</td></tr>
<tr><td>0E</td><td></td><td>Number</td><td>Attach a device or file, with a time out value in the binary data item. (Multiples of 100 msec).
Time out zero is allowed; then control is immediately given back to the task which issued the request. Statuscode indicates whether the device or file is attached.</td></tr>
</table>

Table of National Character Variations

NCV	Countries	-Character Codes									
		Upper case					Lower case				
		/23	/40	/5B	/5C	/5D	/60	/7B	/7C	/7D	/7E
0	Great Britain, Belgium Netherlands	£	@	[	\	]	`	{	|	}	~
1	Germany, Luxemburg, Austria, Switzerland	#	§	Ä	Ö	Ü	`	ä	ö	ü	β
2	France, Switzerland, Belgium, Luxemburg	£	à	°	Ç	§	`	é	ù	ù	¨
3	Spain, Argentina, Venzuela	£	@	[	Ñ	]	`	{	ñ	}	~
4	Italy, Switzerland	£	§	°	C	E	ù	à	ò	è	ì
5	Sweden, Finland	#	É	Ä	Ő	Ȧ	é	ä	ö	ȧ	~
6	Denmark, Norway (1)	£	@	Æ	Ø	Å	`	œ	ø	å	~
7	Portugal, Brazil	£	@	Ã	Ç	Õ	`	ã	ç	õ	~
8	Yugoslavia	£	Ž	Ć	Č	Š	ž	ć	č	š	~
9	USA, Canada, Australia	#	@	[	\	]	`	{	|	}	~
10	Denmark Norway (2)	#	É	Æ	Ø	Å	é	œ	ø	å	ˆ

Note : Use of a lower case character code when Upper Case only has been selected via the DSC1 instruction will result in the equivalent upper case character being printed.

Contol code	Device type	Data-item contains:	Significance
0F		Zero	Detach a device or file. Time out value must be zero.
10	DL	Logical record	The data record and belonging index records are deleted. (The deleted data file records will not be re-used in this release). The index file record is only deleted when data-management has read the data file record correctly.
00	TV	Index value	With this instruction the previously defined parameter table is transferred to the printer. The table has been set up during system generation or by DSC2 with control code X'11'. The data item must contain the index value pointing to the required parameter table. When the document is positioned, new parameters cannot be transferred until the document has been released. If any of the parameters have an illegal value the station is not opened and the instruction is completed with bit 0 set in the status code. This bit is also set if the station is already open and the document has been positioned.

DSC2 *Data set control two* DSC2

Syntax: [statement-identifier] ␣ DSC2 ␣ [.NW,] data-set-identifier,
{control value / equate-identifier} data-item-identifier-1, data-item-identifier-2,
size-identifier

Type: I/O instruction

Description: This statement is used to control a data set referenced by data-set-identifier, which is currently only the teller terminal printer PTS6371.
The kind of control is specified by the control value, which currently can only be X'11'.
.NW indicates that the no wait option is required.
Data-item-identifier-1 refers to a binary or decimal data item containing control information to be passed to the device.
Data-item-identifier-2 refers to a string data item containing the buffer information.
Size identifier refers to a binary data item containing the number of characters to be transferred from the buffer.

Condition register:

= 0 if I/O successful	(OK)
= 2 if Error	(ERR)
= 3 if Begin or end of device	(BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK		ERR	BEOD	$\overline{\text{OK}}$		$\overline{\text{ERR}}$	uncon-ditional

Example: DSC2 DSTP, SDOC, CONTR, BUFF, SIZE

Intermediate code format:

	0			3	4			7
Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	data-set-identifier						
operand-2	control value							
operand-3	data-item-identifier-1							
operand-4	data-item-identifier-2							
operand-5	size-identifier							

Bytes 1 and 2 are filled by the system.
Byte 2 contains a reference to an external system routine.
W is the wait bit.
W=0 no wait
W=1 wait

operand-1 is a reference to the relevant data set
10/100 refers to the first data set.
operand-2 is a hexadecimal integer, which corresponds with the control code.
operand-3 is a reference to a binary data item
operand-4 is a reference to string data item

Control code X'11' Significance

This instruction is used to define the print layout and size of a document, by supplying a set of parameters describing the document. The number of sets is specified during system generation. The different document parameter sets are held in a table in the system, and can be referenced by an index having a starting value of zero for the first entry.

The first entry in the table is supplied with a set of standard parameters for A4 unfolded documents, which may be used if required. These are shown in a table at the end of this instruction description.

These parameters are included during system generation, and this instruction is only used to redefine a parameter set during application running, where a correction or change is necessary.

The number of characters to be transferred must be 14, 18 or 22, depending on the length of the parameter set to be replaced in the table (see below).

Data item identifier-1 refers to a binary data item, which must contain the index value for the required table entry.

Data item identifier-2 refers to a string data item which must contain the set of parameters to be replaced in the table.

All parameters must be supplied in ISO-7 code format.

If any of the parameters are missing or have an incorrect value, the request is completed with CR= 2 (Error), and the table in the system is not updated.

Parameter table entries :

Parameter Type	Length in bytes	Range	Unit
DT	1	0–3	
TO	1	0–9	10s
LS	2	06,10,12,15	1/60"
NL	2	01–99	
BL	2	14–99	1/60"
MA	2	01–80	1/60"
MF	1	1–7	1/60"
LM	1	0,1	
CM	1	0,1	
HP	1	0,1	
UE	2	15–82	1/5",1/10"
BE	2	00,24–99	1/60"
DW/UL	2	40–97/01–40	1/60"/–
CW	2	00–99	1/60"

DSC2 *Continued* DSC2

If DT=0, parameters UE onwards are not required.
If DT=1, parameters DW onwards are not required.
If DT=2 or 3, all parameters are required.

DT: Document type.
0 = Unfolded sheet document with a minimum size of 50 x 110mm. If this type of document is used, a simplified method of positioning is carried out, but this is not as accurate as the method used for other types. When using documents with a height of less than 75mm, this is the only type allowed.
1 = Unfolded documents in general with a minimum size of 75 x 100mm. This is the normal type used for unfolded documents.
2 = Vertically folded (passbook).
3 = Horizontally folded (passbook).

Note that it is possible to print folded documents using DT = 0 or 1, but is this case the positioning is less accurate, and it is the responsibility of the application to see that printing is not performed on the fold. In the case of vertically folded documents, this means that each complete line must be written with two EDWRT or WRITE instructions to ensure that the print head is lifted over the centre fold.

TO: Timeout.
0 = No timeout for document insert.
1–9 = the timeout required in multiples of 10 seconds. If used, the position document – will complete with bit 10 in the return code if no document has been inserted within the specified time.

LS: Line spacing. The distance between two lines, expressed in units of 1/60" (0.423mm).
6 = 10 lines/inch.
10 = 6 lines/inch.
12 = 5 lines/inch.
15 = 4 lines/inch.

NL: Number of lines. The number of evenly spaced lines on the document. Note that, for horizontally folded documents, the area near the fold is treated with the CW parameter (see below). The upper limits of this parameter for different document types and line spacings are as follows:

	Document type		
Line spacing	0,1	2*	3
15	44	25	32
12	55	31	40
10	69	37	48
6	99	61	80

* It is possible to have the same maximum limit on type 2 documents as for type 3, providing the document is thin and folds easily; this will have to be tested before deciding on the parameter to be used.

BL: Bottom Line. The distance between the bottom of the document and the bottom line, expressed in units of 1/60" (0.423mm). This value must be in the range 14–99 inclusive, which means that the bottom line may be placed between 6 and 42mm from the bottom of the document. See diagrams at the end of this description for clarification.

MA: Margin. The width of the margin expressed in units of 1/10".

MF: Margin fine. The width of the fine margin expressed in units of 1/60". The sum of MA + MF is the distance between the right-hand edge of the document and the margin (left or right). The rightmost position of a right margin is 8mm from the right-hand edge of the document, and this corresponds to the sum MA + MF = 1. The leftmost position of a left margin is 206.2mm from the right-hand edge of the document, and this corresponds to MA = 80, MF = 7. The left margin must not, however, be placed closer than 3mm to the left-hand edge of the document.

LM: Left margin.
0 = Print with right margin.
1 = Print with left margin.

CM: Critical margin.
0 = No critical margin.
1 = Critical margin. This must be set if the margin or any text is intended to be positioned closer than 6mm from the edges of the document. In this case, the print speed is reduced near the edges to prevent the head overrunning the document edges. Note that for document type 0, it may not be necessary to set this parameter to one, even if printing close to the edge; this will have to be tested in each case.

HP: High pressure.
0 = Normal print pressure.
1 = High print pressure, primarily intended for printing on multiple sets of forms.

UE: Upper edge. This is not significant for document type 0.
For document type 1, this is the distance between the bottom of the document and the true upper edge, expressed in units of 1/5" (5.08mm). As the limits for this value are 15–63, this means that a document with a height of 75mm to 316mm can be used. See also the diagram at the end of this description for further clarification.
For document type 2, this is the distance between the bottom of the document and the upper edge of the pages, expressed in units of 1/10" (2.54mm). The normal limits for this value are 25–82, but note that the distance between the bottom and upper edge must not be less than 60mm, and the total height of the document must not be more than 210mm.

For document type 3, this is the distance between the bottom of the pages of the document and the upper edge of the pages, expressed in units of 1/10" (2.54mm). The normal limits for this value are 48–82, but note that the minimum distance between the bottom and upper edges is 120mm, and the total height of the document must not exceed 210mm. Horizontally folded documents with a distance of less than 120mm from bottom to upper edge will need to be tested specially, to check that the quality of the print is good enough. The absolute lower limit for this parameter and this document type is 40. This parameter is required to ensure that the print head is lifted as the physical edges of the pages could otherwise jam in the grasp mechanism.

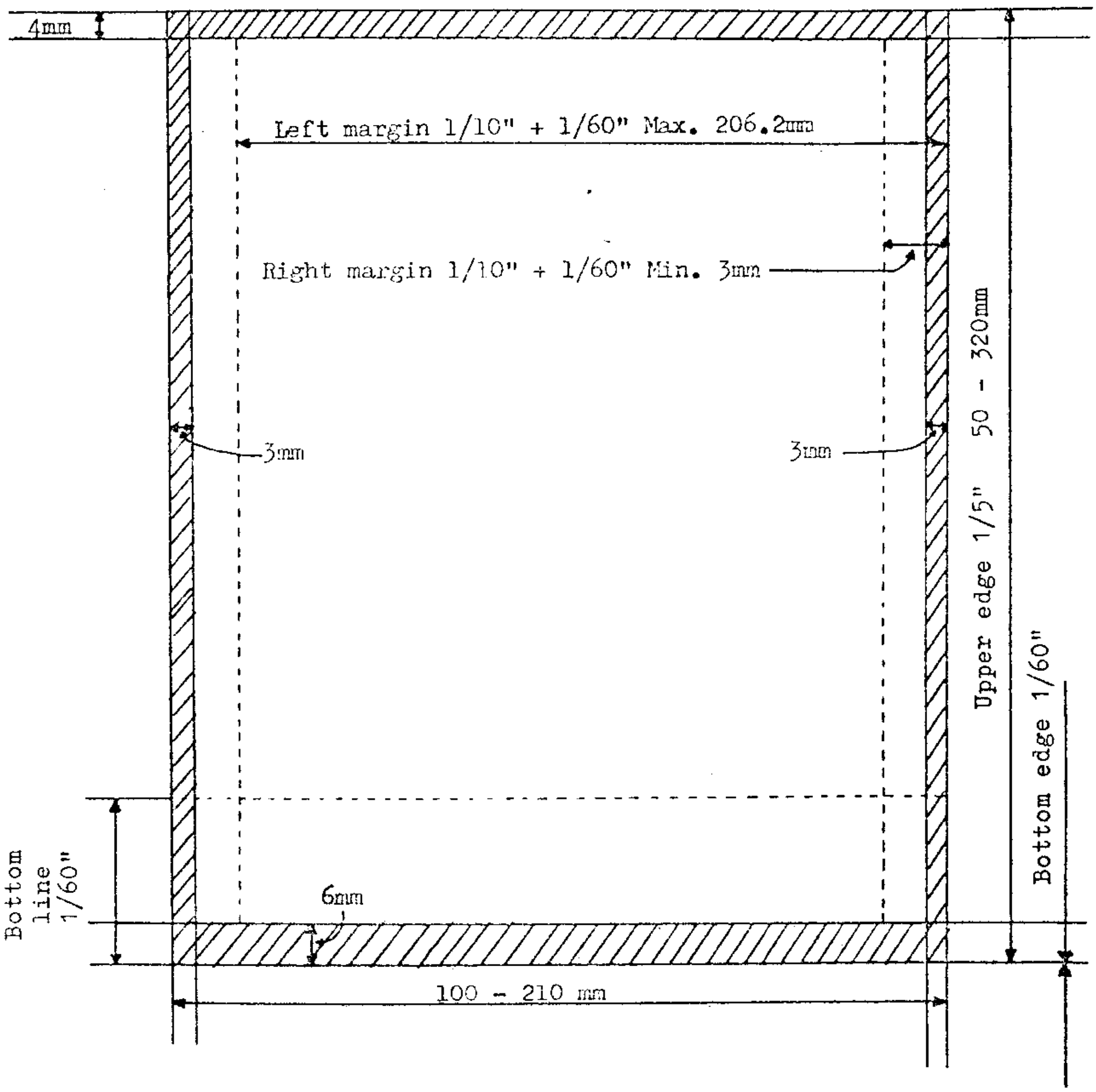
BE: Bottom edge. This parameter is not significant for document type 0. For all other document types, this is the distance between the bottom of the document and the bottom of the pages, expressed in units of 1/60" (0.423 mm). See the diagram at the end of this description for further clarification. The limits of this value are 24–99 or zero, which means that the bottom of the pages must be placed 10–42 mm from, or in line with the bottom of the document. This is normally set to zero for document type 1.
This parameter is required to ensure that the print head is lifted as the physical edges of the pages could otherwise jam in the grasp mechanism.

DW: Document width. This is only significant for document type 2, and is the width of the document in units of 1/10" (2.54mm).

UL: Upper lines. This is only significant for document type 3, and is the number of lines on the upper portion of a horizontally folded document.

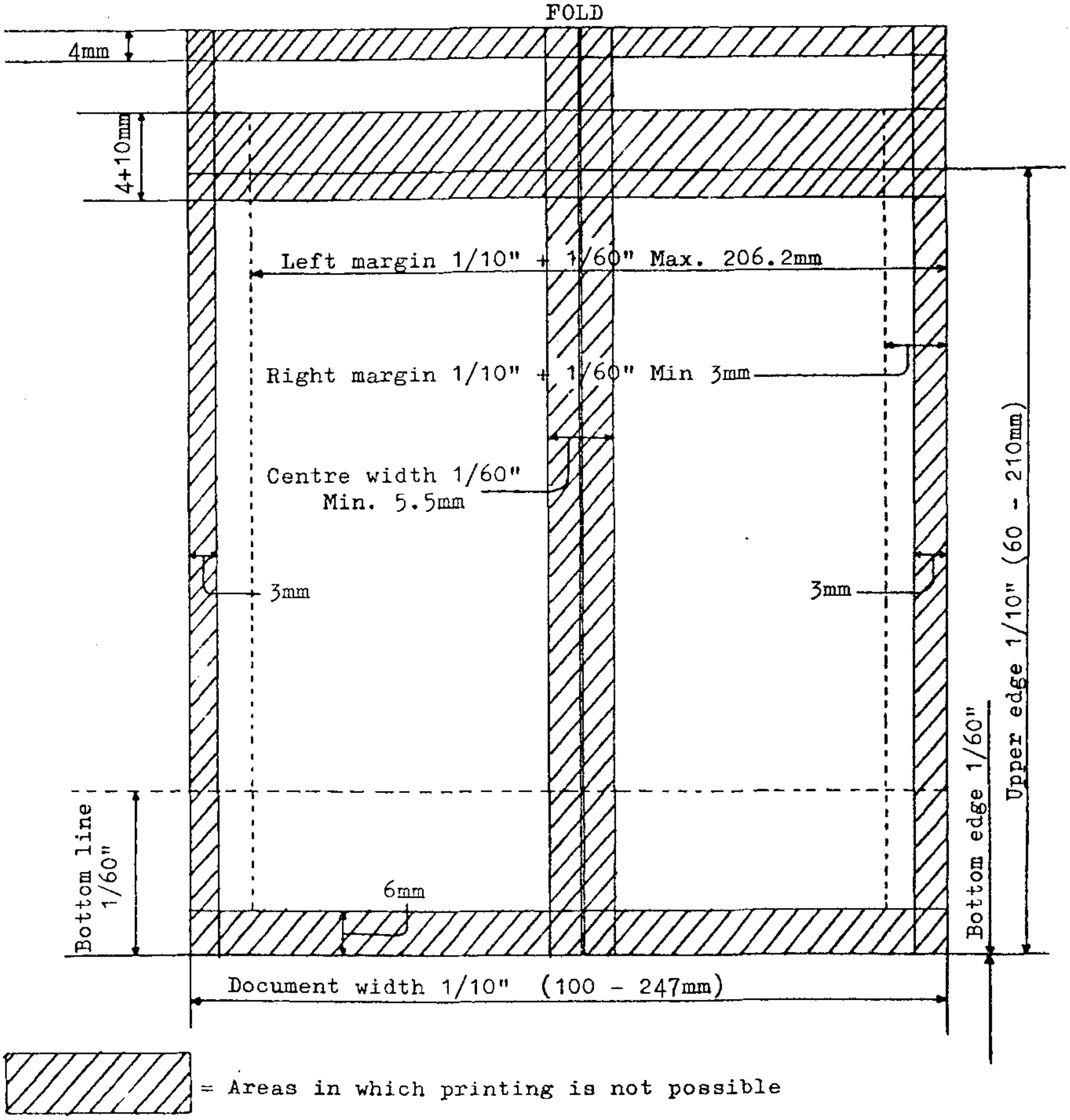
CW: Centre width. This is not significant for document types 0 and 1.
For document type 2, this is the width across the fold on vertically folded documents, where the print head must be released as no printing is permitted, expressed in units 1/60" (0.423mm).
For document type 3, this is the distance from the bottom line on the upper portion of a horizontally folded document to the first line on the lower portion of the document, expressed in units of 1/60" (0.423mm).

Diagram of parameters for document types 0, 1 (Unfolded document)


4mm
Left margin 1/10" + 1/60" Max. 206.2mm
Right margin 1/10" + 1/60" Min. 3mm
3mm
3mm
Upper edge 1/5" 50 - 320mm
Bottom edge 1/60"
Bottom line 1/60"
6mm
100 - 210 mm

▨ = Areas in which printing is not possible

Diagram of parameters for document type 2 (Vertically folded)


FOLD
4mm
4+10mm
Left margin 1/10" + 1/60" Max. 206.2mm
Right margin 1/10" + 1/60" Min 3mm
Centre width 1/60"
Min. 5.5mm
3mm
3mm
Upper edge 1/10" (60 - 210mm)
Bottom edge 1/60"
Bottom line 1/60"
6mm
Document width 1/10" (100 - 247mm)
= Areas in which printing is not possible

DSC2 *Continued* DSC2

Table of National Character Variations

NCV	Countries	Character Codes									
		Upper case					Lower case				
		/23	/40	/5B	/5C	/5D	/60	/7B	/7C	/7D	/7E
0	Great Britain, Belgium Netherlands	£	@	[	\	]	`	{	|	}	~
1	Germany, Luxemburg, Austria, Switzerland	#	§	Ä	Ö	Ü	`	ä	ö	ü	β
2	France, Switzerland, Belgium, Luxemburg	£	à	°	Ç	§	`	é	ù	ù	¨
3	Spain, Argentina, Venzuela	£	@	[	Ñ	]	`	{	ñ	}	~
4	Italy, Switzerland	£	§	°	C	E	ù	à	ò	ė	ì
5	Sweden, Finland	#	E	Ä	Ö	Ȧ	é	ä	ö	ȧ	~
6	Denmark, Norway (1)	£	@	Æ	Ø	Å	`	æ	ø	å	~
7	Portugal, Brazil	£	@	Ã	Ç	Õ	`	ã	ç	õ	~
8	Yugoslavia	£	Ž	Ć	Č	Š	ž	ć	č	š	~
9	USA, Canada, Australia	#	@	[	\	]	`	{	|	}	~
10	Denmark Norway (2)	#	É	Æ	Ø	Å	é	æ	ø	å	▴

Note : Use of a lower case character code when Upper Case only has been selected via the DSC1 instruction will result in the equivalent upper case character being printed.