PHILIPS

PTS 6800 TERMINAL SYSTEM

User Library

CREDIT
Programmer's Guide

Module M09

Data
Systems

Date : November 1977

Code : 5122 993 39931

MANUAL STATUS SURVEY

Module M09 "CREDIT PROGRAMMER'S GUIDE"

This issue comprises following updates :

– U1.39931.1079 (October 1979; complete revision for CREDIT Release 4.1)

CONTENTS

2. CREDIT FEATURES

CREDIT is a product area designed real time computer language, it has been designed specifically for programming real time applications on the PTS. Product area designed languages such as CREDIT have several advantages over general pupose languages such as FORTRAN. Some of the more important features of CREDIT are given below.

2.1 Maximize machine useage

CREDIT is an interpretive language, it does not hold a "core image" of the application for each user, just a pointer to the next statement to be obeyed and a set of variables for each user.

2.2 Wide range of Input/Output commands

An important consideration in the design and writing of any application must be that the final item is "user friendly", otherwise it may never be accepted by the users, be they bank clerks or managers. To help the programmer produce a "user friendly" application, CREDIT offers a number of data set control, input and output commands and routines to handle such actions as the printing of pass books, the displaying of signatures on a plasma display, or option lists on a visual display unit.

2.3 Programmed by terminal class, not work station

CREDIT allows the programmer to define different classes of terminals to handle different functions within the application. For example one terminal class could be for the bank tellers, one for the foreign exchange desk and one further terminal class for the manager. A group of similarly configured work positions, handling the same types of transaction, is known as a terminal class. Because all work positions in a terminal class handle the same type of transaction the same program code is used for each of the work stations. There can be up to sixteen terminal classes in one application.

2.4 Work blocks for storage and communication

Within the CREDIT terminal class it is possible to have a range of different data structures. Data items can be associated with a work station to record, for, example cash handled at that point; or they could be associated with a system user in which case the user would have to enter a recognition code; or they could be common to all users, holding the current date and time for example.

2.5 Multi-task system

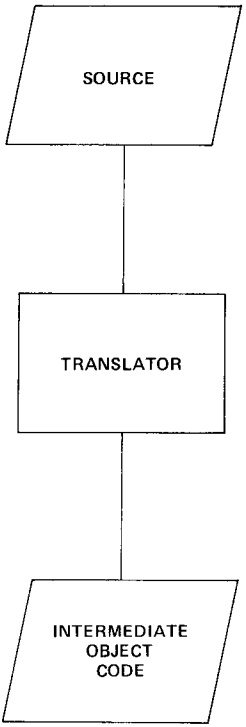
Each work station within a terminal class is regarded as a separate task; the work station may be a keyboard and display with an operator entering information, or it may be controlling data communication with a remote processor. Every time data is received from a work station the Terminal Operating System Software (TOSS) Monitor activates the appropriate task. Hence, several tasks will seemingly be active at the same point in time, however, it is only possible for the computer to carry out one activity at one point in time and this problem is overcome by the TOSS Monitor, which schedules tasks such that all appear to be run simultaneously.

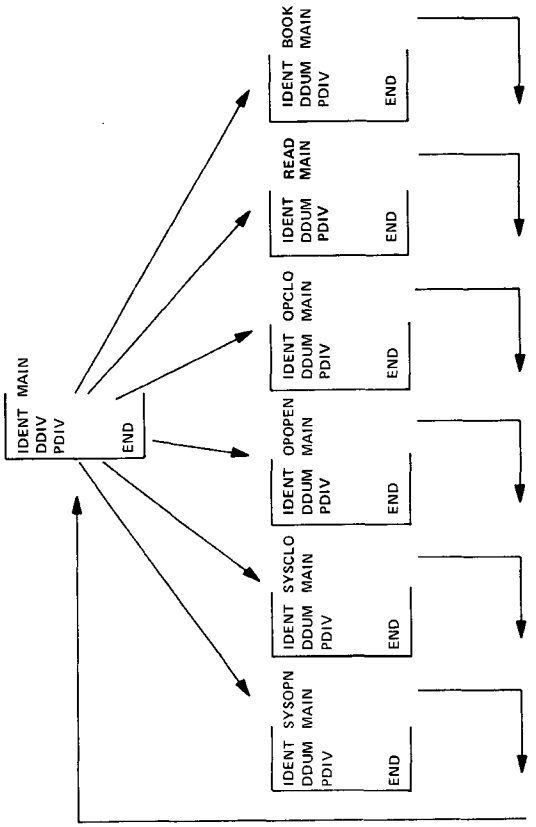
2.6 Specialised command set

The CREDIT command set is specifically designed for handling real time applications; special commands exist to control the printing of bank pass books etc., handling data communication with a remote processor or controlling the layouts on a visual display screen.

3. **CREDIT** PROGRAM STRUCTURE

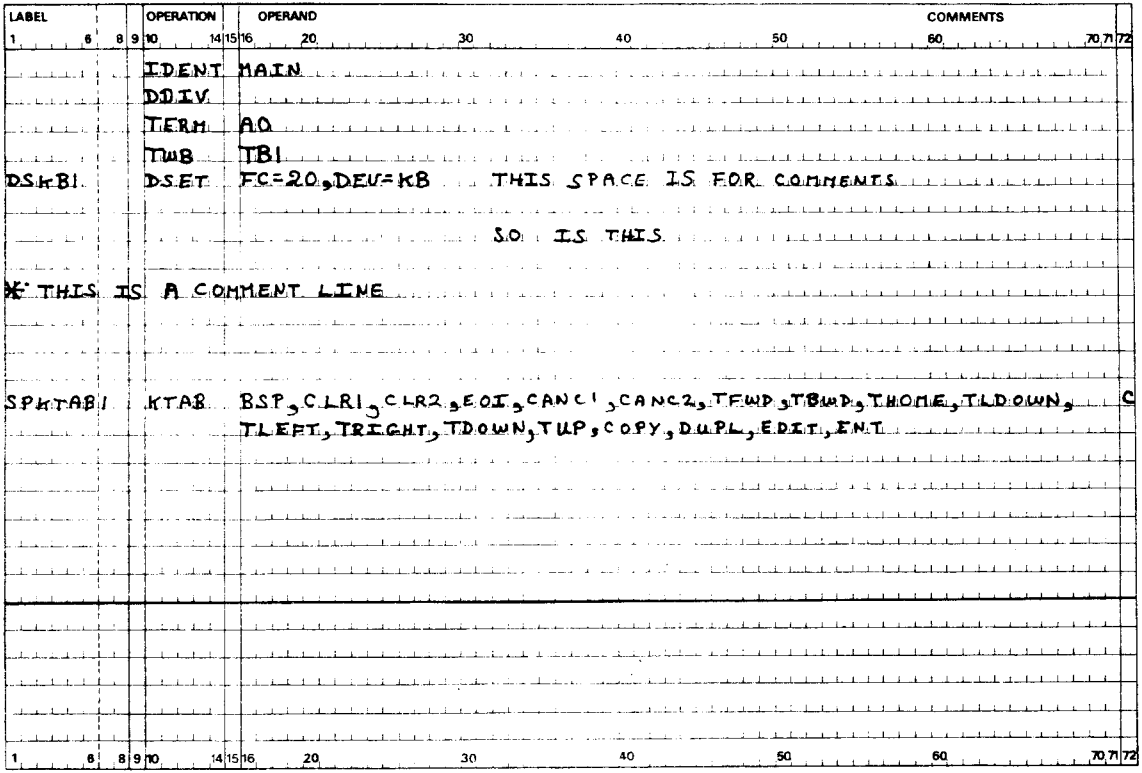
A CREDIT program consists of a number of statements. A statement may be a directive, declaration or instruction. Directives describe the module framework to the CREDIT translator, the translator is a special program which converts the application program into a form the machine can use, this output being called 'object code'. Declarations specify the type, length and use of all the variables, constants and tables used in the application and must always be located in the main module. Instructions direct the input, processing and output of information. They specify the actions to be carried out by the computer, and direct the sequence of events.


SOURCE
TRANSLATOR
INTERMEDIATE
OBJECT
CODE



MODULAR STRUCTURE WITH A GLOBAL DATA DIVISION

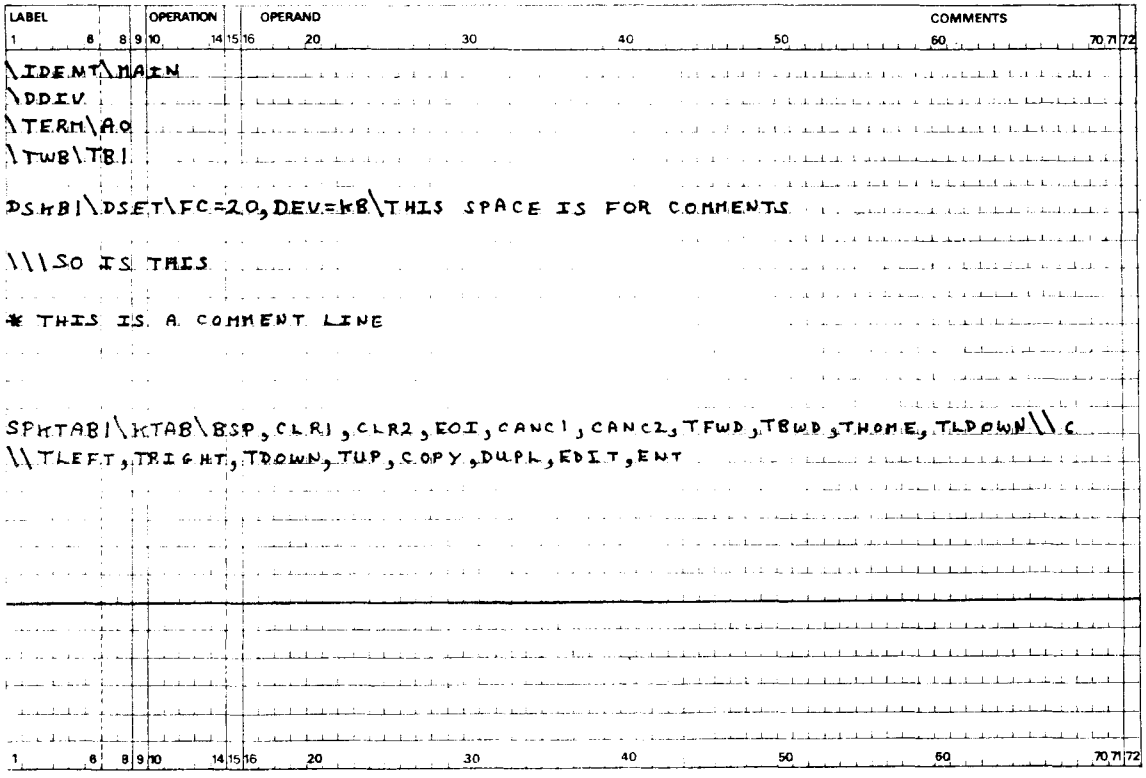
The program is written as one or more modules; with the PTS system, as with many others, it is advantageous to have one module performing one specific activity, as this leads to more efficient design, writing, testing and subsequent maintenance of the application.



```
LABEL          OPERATION  OPERAND                                                    COMMENTS
1      6   8 9 10    14 15 16  20        30        40        50        60        70,71 72
                  IDENT  MAIN
                  DDIV
                  TERM   A0
                  TWB    TB1
DSKB1             DSET   FC=20,DEV=KB      THIS SPACE IS FOR COMMENTS
                                                 SO IS THIS
*  THIS IS A COMMENT LINE
SPKTAB1           KTAB   BSP,CLR1,CLR2,EOI,CANC1,CANC2,TFWD,TBWD,THOME,TLDOWN,    C
                         TLEFT,TRIGHT,TDOWN,TUP,COPY,DUPL,EDIT,ENT
```

A CREDIT source program can be read into the PTS system using one of the following source input devices; cassettes or console typewriter for free format input, and cards for fixed format input. Regardless of the input device used, the source program must have follow the rules described below and shown on the attached coding sheet of examples.

A source line input to the translator is treated as an 80 character card image; if a free format input device is used then each record can only contain one source statement. Records longer then eighty characters will be truncated, while any shorter will be filled with spaces to pad them to eighty characters.

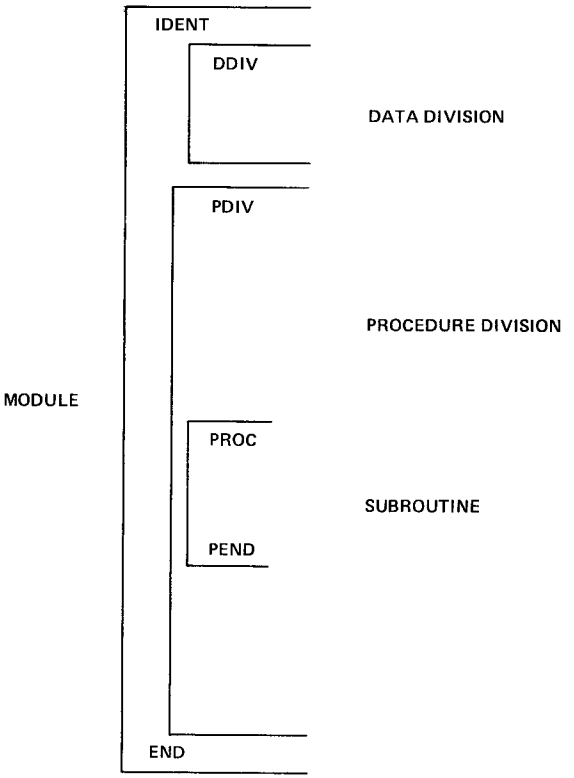

LABEL
OPERATION
OPERAND
COMMENTS
1 6 8 9 10 14 15 16 20 30 40 50 60 70 71 72
\IDEMT\MAIN
\DDIV
\TERM\A0
\TWB\TB1
DSMB1\DSET\FC=20,DEV=MB\THIS SPACE IS FOR COMMENTS
\\\SO IS THIS
* THIS IS A COMMENT LINE
SPMTAB1\MTAB\BSP,CLR1,CLR2,EOI,CANC1,CANC2,TFWD,TBWD,THOME,TLDOWN\\C
\\TLEFT,TRIGHT,TDOWN,TUP,COPY,DUPL,EDIT,ENT

The source line is divided into four fields or zones:- label field, operation field, operand field and comment field. The label, operation and operand field are separated by a tabulation character (\) or at least one space. The label field begins in column one. The operand field can extend to column 71. If the operation and operand fields are blank to column 30 the rest of the record is treated as a comment. Columns 73 through 80 are ignored by the translator. If an asterisk is present in column one then the entire record will be treated as a comment. Continuation of a line is denoted by a 'C' in column 72 if card input is being used, or in the case of free format input by (\\C) two tab symbols followed by 'C', in the continuation lines the label and operation field must be left blank.

3.1 DIRECTIVES

Directives enable the application programmer to pass information to the CREDIT translator. The directives do not occupy any 'core' at run time. There are six categories of directives:-

- Structure
- Linkage
- Listing
- Equate
- Parameter
- Options



FRAMEWORK OF A CREDIT MODULE (WITH DATA DIVISION)

3.1.1 Structure directives

The framework of a CREDIT module is formed from the directives IDENT, DDIV, PDIV, PROC, PEND, INCLUDE and END. An example of their use is given below.

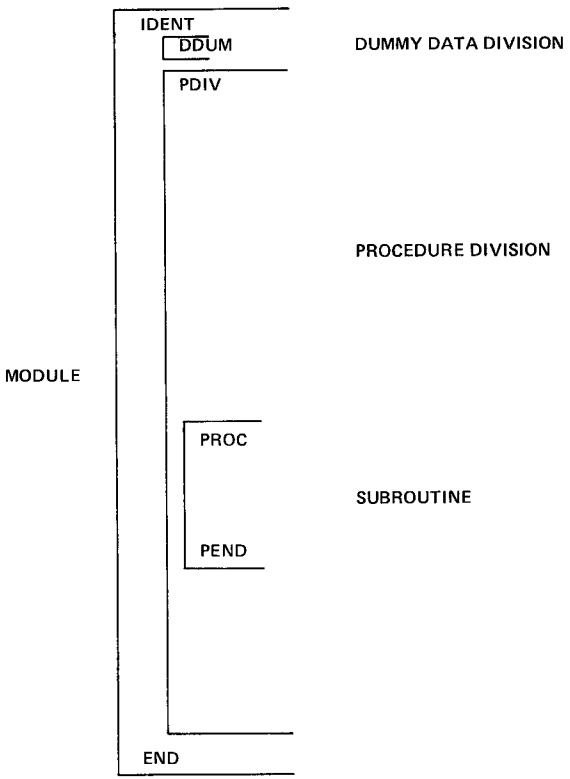
IDENT	Must be the first statement
DDIV(or DDUM)	Start of the data division

The data division contains declarations which define the type, length and value of data items used as operands in the program, together with declarations which define the interface between the applications program and the PTS System.

PDIV	Start of the procedure division

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.

PROC	Start of subroutine instructions
PEND	End of subroutine instructions
	Several subroutines may exist in one module.
INCLUDE	The contents of a source module are included in this module at this point
END	Must be the last statement



FRAMEWORK OF A CREDIT MODULE (WITH DUMMY DATA DIVISION)

3.1.1.1 Main program structure directives

The IDENT and END directives define the start and end of a module, and must be the first and last statements respectively of a module. The DDIV directive defines the start of the data division, and must be the second statement of the module containing the data division. The dummy data division directive (DDUM) is used in all other modules in place of the DDIV directive, and it refers to the IDENT of the module containing the required DDIV. The PDIV directive defines the start of the procedure division.

3.1.1.2 Subroutine structure directives

The PROC and PEND directives define the start and end of each subroutine. The IDENT, DDIV (or DDUM), PDIV and END directives can only appear once in each module; the PROC and PEND directives must be repeated for each subroutine present. However one subroutine can not be physically embedded within another, that is two or more PROC directives can not occur without an intervening PEND.

Example:

VALID

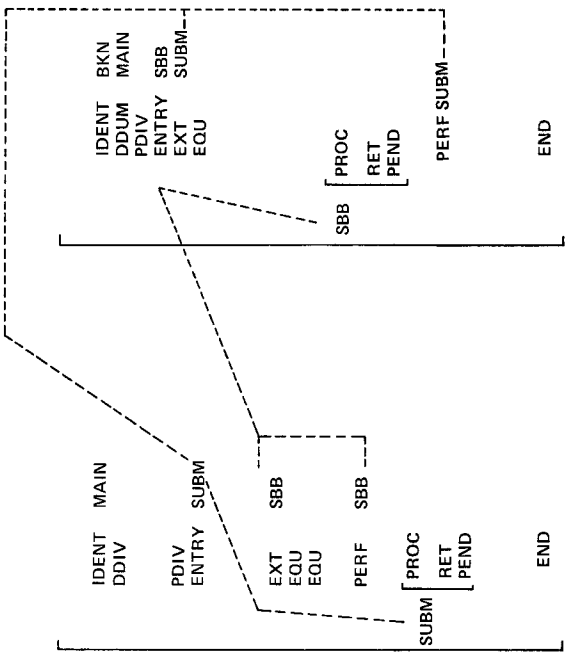
```
S1      PROC
        PEND
S2      PROC
        PEND
```

INVALID

```
S1      PROC
S2      PROC
        PEND
        PEND
```

The table below gives the structure directives and the page on which they are described in M04.

DIRECTIVE	PAGE IN M04
DDIV	1.2.3
DDUM	1.2.4
END	1.2.6
IDENT	1.2.10
INCLUDE	1.2.11
PDIV	1.2.17
PEND	1.2.17
PROC	1.2.21



ENTRY/EXTERNAL REFERENCES

3.1.2 Linkage directives

Linkage to external modules

CREDIT modules which have to be linked into an application program may contain references to statements or subroutines in other modules. In order to achieve the correct linkages, entry points in this module and external references to other modules must be specified. The ENTRY and EXT directives are used for this purpose. They must be written in the procedure division.

Thus, in order for the references to be correctly handled, a module referring to a statement-identifier in another module must contain the EXT directive to specify that the reference is not in this module, and this must be paired with an ENTRY directive in the other module.

Start points

There must be at least one START directive for the entire application. When the system is started (i.e. the TOSS Monitor is loaded and the application program begins execution) tasks are activated as specified in the configuration data to be studied later. The tasks are activated at the start points specified in the START directives of the relevent terminal classes. The START directive(s) must be written in the data division and must be specified as entry points (ENTRY) in the procedure division (PDIV).

If more than one START directive appears in a terminal class, only the first start point will be activated when the system is started; the other points will be held pending and will be activated only after the first task has executed an EXIT instruction.

Error control with memory management, swappable work blocks or overlay

If memory management is being used and a REENTER point has been defined for handling disk errors, then this must also be declared as an ENTRY point.

The table below gives the linkage directives and the pages on which they are described in M04.

DIRECTIVE	PAGE IN M04
ENTRY	1.2.7
EXT	1.2.9
REENTER	1.2.22
START	1.2.23

3.1.3 Listing directives

These directives are used to control the printing of the application listing at translation time. The available directives are:-

- LIST
- NLIST
- EJECT

EJECT when encountered issues a form feed to the printer, NLIST stops the production of the program listing, LIST causes the listing to recommence.

The table below gives the listing directives and the pages on which they are described in M04.

DIRECTIVE	PAGE IN M04
LIST	1.2.7
NLIST	1.2.9
EJECT	1.2.22

3.1.4 Options directive

This directive, if required, must be located immediately after the DDIV or DDUM directive and controls the following items:-

- Lines per page for translator listings
- One or two byte addressing for data items
- One or two byte addressing for literal constants
- One or two byte addressing for keytables
- One or two byte addressing for pictures
- One or two byte addressing for format lists
- One or two byte addressing data sets
- Number of entries in work blocks

A one byte addressing system allows only 16 entries per work block whereas a two byte system would permit up to 255 entries. With one byte addressing up to 255 literal constants, keytables, pictures and format lists are allowed; if two byte addressing were used then there could be up to 32767.

The lines directive is overruled if the number of lines per page is given when entering the program development sytem (DOS-PTS).

The various options can be written in any order.

The format of this directive is:-

```
OPTNS     {LINES=decimal number,}{LITADR=decimal number,}{ADRMOD=decimal
                                                          number}
```

The LITADR option is followed by a four digit decimal number composed of one's or two's, a one representing one byte addressing a two, two byte addressing. The first digit of the number is for literal constants, the second keytables, the third pictures and the fouth format lists.

The address mode option (ADRMOD) is used to specify one or two byte addressing for data items, literal constants, data sets, keytables, format lists and pictures; the valid forms of ADRMOD are shown below:-

ADRMOD=1 one byte addressing is to be used (default).

ADRMOD=2 two byte addressing is to be used; LITADR will be set by the translator to 2222.

Example 1

```
OPTNS     LINES=72,LITADR=2212
```

In example 1 there will be seventy two lines per page on the program listings, and two byte addressing will be used for literal constants, keytables and format lists; but a one byte addressing system is to be used for pictures.

Example 2

```
    OPTNS    ADRMOD=2
```

In example 2 two byte addressing is to be used, permitting more workblock entries (data items), two byte addressing will also be used for the literals. Literals, keytables, format lists, pictures and data sets are described in sections 3.2.8, 6.2.3, 6.5 and 3.2.6 respectively.

3.1.5 Equate directive

The EQU directive is used to set up constants with mnemonic names; the prorammer uses the name when writing the instructions, and when the program is translated the constant is substituted for the name. The EQU directives can be located anywhere in the procedure division after the ENTRY and EXT directives. The maximum value that can be held in an equate directive is 255.

The directive format is:-

mnemonic-name EQU value-expression

eg

```
    BSP    EQU    X'09'          BSP to be replaced ty Hex. value 9
    NDBS   EQU    BSP            NDBS to be replaced by contents of BSP
    CHA    EQU    X'40'          CHA to be replaced ty Hex. value 40
    NGP    EQU    CHA+1          NGP to be replaced ty Hex. value 41
```

DIRECTIVE	PAGE IN MO4
EQU	1.2.8

3.1.6 Parameter directives

These are special directives for use when passing format lists, keytables, literals and parameters to subroutines. There are four directives used within the subroutines:-

- PFRMT for format lists
- PLIT for literals
- PKTAB for keytables.
- PLIST for subroutine parameters

They are required when ADRMOD is set to two, or if the formal parameter is not preceeded by a $ sign, and are located after the directive PROC, see below:-

```
SUBF    PROC    FORM1       (ADRMOD=2)
        <opt>   FORM1

SUBF    PROC    $FORM1      (ADRMOD=1)

SUBF    PROC    FORM1       (ADRMOD=1)
        <opt>   FORM1
```

<opt> is either PFRMT, PLIT or PKTAB.

PLIST Actual parameter list

This is used to pass parameters to subroutines that have been activated using the indexed perform (PERFI) instruction.

The handling of subroutines, and the syntax of parameter passing is described in detail in section 5.

DIRECTIVE	PAGE IN M04
PFRMT	1.2.18
PLIT	1.2.20
PKTAB	1.2.19
PLIST	1.4.210

3.2 Data division

3.2.1 Overview

The DDIV contains declarations which define the type, length and value of data items used by the program, together with those declarations which define the interface between the application and TOSS Monitor. The basic layout is shown below, and will be described in detail later.

```
          DDIV                    Directive for start of data division
          TERM      TO            Terminal class identifier
          TWB       TB1           Terminal work block TB1
DSKB1     DSET      FC=20,DEV=KB  Definition for keyboard input
          START     S1            First start point for this terminal class
          START     S2            Next start point for this terminal class
.....................................
Work block declarations are located
in this section
.....................................

          PDIV                    Start of procedure division
          ENTRY     S1            Entry point S1 is in this module
          ENTRY     S2            Entry point S2 is also in this module
          EXT       SCREEN        Externally held subroutine
KEY       EQU       D'56'
```

Explanation of the above example

i DDIV - this indicates the start of the data division.

ii TERM TO - the terminal class identifier; TO is the name of the terminal class, as described in section 3.2.3.

iii TWB TB1 - TB1 is to be used as a terminal work block for this terminal class. A description of work blocks is given in section 3.2.5.

iv DSKB1 DSET FC=20,DEV=KB - assigns a dataset device to be used by this terminal class, which will be referred to in the program as DSKB1. It has a file code of 20 and is a keyboard device. The DSET command is described in detail in section 3.2.6.

START- This gives the program entry point where execution will commence. The directive is described in section 3.2.4.

DATA DIVISION (1)

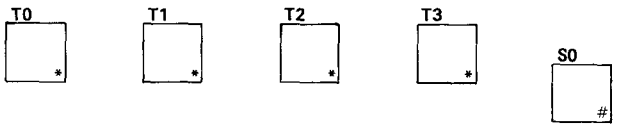
```
                IDENT   MAIN
                DDIV
                TERM    T0
                CWB     CB1
                CWB     CB2
                TWB     TB1
                TWB     TB2
                START   G0
DSKB            DSET    FC=20,DEV=KB
DSJT            DSET    FC=34,DEV=JT,BUFL=80
DSVO            DSET    FC=30,DEV=TP,BUFL=80

CB1             BLK
RDLA            BIN     X'0080'
ADDLA           BIN     X'0001'
INDX            BIN
DATE            BCD     10D'0'
WKSTR           STRG    5

CB2             BLK

TB1             BLK

TB2             BLK

                PDIV
```

```
vi    TB1       BLK - this is the start of the declarations which form work
                      block TB1, work block definitions must be located at
                      the end of the terminal class definitions

iii             PDIV - start of the procedure division

viii            ENTRY S1 - the entry point S1 will be located in this module

ix              EXT  SCREEN - this is reference to an externally held routine

x     KEY       EQU  D'56' - a constant is set up with the decimal value
                             56. The maximum value for a constant is 255.
```

3.2.2 Structure of the data division

The data division is divided into two sections, the first contains the terminal class definitions and the second defines the data items that make up the work blocks. An example of a data division is given below, with two terminal classes; note that the terminal classes must all be defined before the work blocks.

```
            DDIV

            TERM      T0              Terminal class identifier
            TWB       TB1             Terminal work block TB1
            CWB       CX1             Common work block
            CWB       CX2             Common  work block
DSKB1       DSET      FC=20,DEV=KB    Definition for keyboard input
DSDY1       DSET      FC=50,DEV=DY,BUFL=120         vdu display
            START     S1              Start point for this terminal class

            TERM      S0              Terminal class identifier
            TWB       TB1             Terminal work block TB1
            CWB       CX1             Common work block
DSKB1       DSET      FC=20,DEV=KB    Definition for keyboard input
DSPRT       DSET      FC=40,DEV=LP,BUFL=240         line printer
            START     S2              Start point for this terminal class
            STACK     128             Stack size for this class

.................................
Work block declarations are located
in this section
.................................

          PDIV
```



# *	MAIN		
# *	INPUT		
# *	PROCESS		
# *	OUTPUT		
*	CURRENT ACCOUNT		
*	SAVINGS ACCOUNT	TERM	T0
		TWB	TB1
		TWB	TB2
#	CURRENCY EXCHANGE	TWB	TB3
		CWB	CB1
*	OPEN ACCOUNT		
*	CLOSE ACCOUNT	TERM	S0
		TWB	TB4
		TWB	TB5
#	CHEQUES	TWB	TB6
		CWB	CB1

* MODULES (PROGRAMS) USED BY TERMINAL CLASS T0

\# MODULES (PROGRAMS) USED BY TERMINAL CLASS S0

TERMINAL CLASS

3.2.3 Terminal class declaration

The TERM declaration identifies the terminal class with a unique two character identifier. This declaration is followed by the relevent work blocks, start points, data set identifiers etc. for the terminal class. A terminal class is defined as a collection of work stations performing similar functions.

Each terminal class has its own specified work blocks, input/output devices, entry and reentry points stack; a terminal class may consist of several work stations. Each work station or task forming the terminal class will have its own copy of the above mentioned items. For example terminal class S0 has a specified stack size of 128 bytes, so each task forming that class will have its own stack 128 bytes in size. The STACK declaration is described in M04.

Each work station in a terminal class is identified with a 'task identifier', which is specified at system configuration time. The first task in terminal class T0 will have a task identifier of T0, the second will have a task identifier of T1, the nth task wil have an identifier of Tn-1.

In the example on page 3.2.6 the task T0 has been configured with four copies, the tasks forming this class have the identifiers T0, T1, T2 and T3.

DECLARATION	PAGE IN M04
STACK	1.3.22
TERM	1.2.26

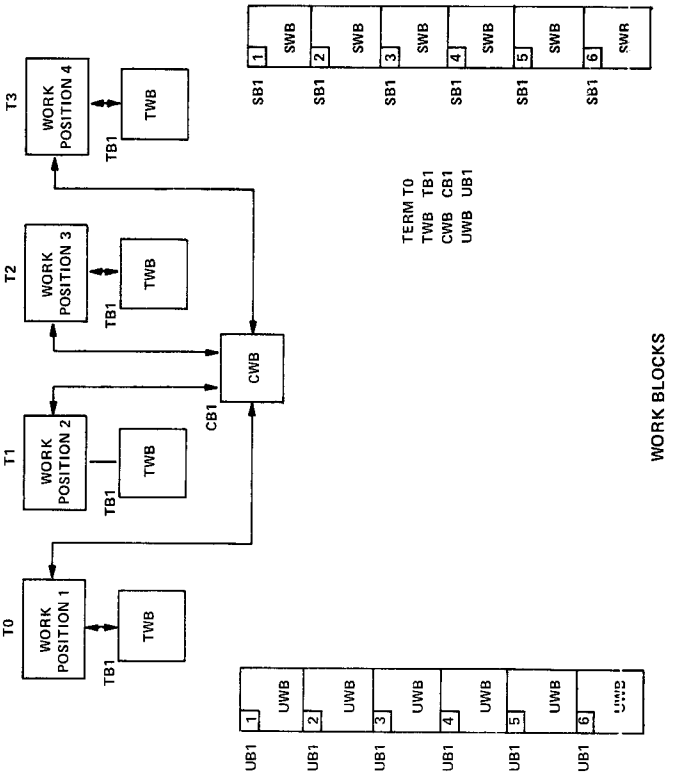
3.2.4 Start directive

The start directive gives the program entry point where program execution will commence for this task. There must be at least one start point in a program.

If a terminal class contains multiple start points the first will be used at the start, subsequent start points only being used when an EXIT is encountered. In the example shown in section 3.2.2, execution will commence at S1 but when an EXIT is encountered execution wil pass to entry point S2.

If a terminal class does not contain a START directive then it can only be invoked by another terminal class with the activate (ACTV) instruction

DECLARATION	PAGE IN M04
START	1.2.23


T0
WORK POSITION 1
TB1
TWB
T1
WORK POSITION 2
TB1
TWB
T2
WORK POSITION 3
TB1
TWB
T3
WORK POSITION 4
TB1
TWB
CB1
CWB
UB1
1 UWB
2 UWB
3 UWB
4 UWB
5 UWB
6 UWB
SB1
1 SWB
2 SWB
3 SWB
4 SWB
5 SWB
6 SWB
TERM T0
TWB TB1
CWB CB1
UWB UB1
WORK BLOCKS

3.2.5 Workblocks

These are used by the programmer to provide areas of store which can be used for input/output buffers and work locations. There are five types of work block:

- Terminal workblocks (TWB)
- Common workblocks (CWB)
- User workblocks (UWB)
- Dummy workblocks (DWB)
- Swappable workblocks (SWB)

There can be a maximum of fifteen work blocks in a terminal class and 16 non-boolean entries in a block, though this can be increased to 256 non-boolean entries in a block if the option ADRMOD=2 is specified in OPTNS directive, as described in section 3.1.4. Each workblock can contain up to sixteen boolean items, as one word is reserved for these data items per block. Non-boolean items occupy one entry in the workblock, except arrays which take up <u>two</u> entries in a work block, <u>unless</u> they are the last item in a work block when they occupy only one entry.

Valid 16 entries

```
TB1      BLK
I1       BOOL
NBIN1    BIN
NBIN2    BIN
ABIN3    BINI      (4)
ABCD4    BCD       5D'0'
ABCD5    BCDI      (12),5D'0'
TBCDX    BCDI      (8),12D'0'
ACCX     BCDI      (99),8D'0'
BRANCH   STRGI     (40),10C
MNGR     STRGI     (40,2),25C
TELNO    STRGI     (40,3),10C
```

Invalid 17 entries

```
TB2      BLK
I1       BOOL
NBIN1    BIN
NBIN2    BIN
ABIN3    BINI      (4)
TELNO    STRGI     (40,3),10C
ABCD5    BCDI      (12),5D'0'
TBCDX    BCDI      (8),12D'0'
ACCX     BCDI      (99),8D'0'
BRANCH   STRGI     (40),10C
MNGR     STRGI     (40,2),25C
ABCD4    BCD       5D'0'
```

The declaration for a terminal block consists of the mnemonic for the block type in the operator field and the name of the block in the operand field, eg:-

```
         TWB       TB1
```

Associated with each work block declaration will be a work block description where each data item forming part of that work block is described eg:-

```
TB1      BLK
CASID    STRG      6C'*'
DEP      BCD       12D'0'
WHDL     BCD       12D'0'
```

3.2.5.1 Terminal workblocks

A terminal class can contain one or more terminal workblock (TWB) definitions, each task within the terminal class having a separate copy of the work blocks. Each terminal work block will contain the data items to be used by that terminal class; the list of terminal work blocks to be used will be located after the terminal identifier (TERM), the work blocks being located after the terminal definitions.

For example:-

```
            TERM    A0
            TWB     TB1
            TWB     TB2
            TERM    B0
            TWB     TB1

TB1         BLK
NAME        STRG    20
ADDR        STRG    30
ACNO        BCD     12
OVFT        BCD     8
BAL         BCD     8
TRAN        STRG    3
FLAG        BCD     2D

TB2         BLK
FL          BOOL
SPPROMPT    BOOL
SPCHANGE    BOOL
SPERCALL    BOOL
SPBINW1     BIN
SPBINW2     BIN
SPBINW3     BIN
SPBINW4     BIN
SPINPUT     STRG    80C' '
SPSTRGW1    STRG    2C' '
```

In the above example the data items making up terminal work block TB1 will be available to terminal classes A0 and B0, but TB2 is only available to terminal class A0. Each task making up the terminal classes will have a separate copy of the data items, and so it is not possible to use terminal workblocks to pass information to, or receive information from other tasks.

IDENTIFIER	PAGE IN M04
TWB	1.3.4

DATA DIVISION (2)

```
            IDENT   MAIN
            DDIV
            TERM    TØ
            CWB     CB1
            TWB     TB1
            START   GØ                          ILLEGAL
DSKB        DSET    FC=20,DEV=KB

            TERM    SØ                  TERM    SØ
            CWB     CB1 ..............  TWB     TB2
            TWB     TB2 ..............  CWB     CB1
            START   SØ GØ               START   SØ GØ
DSKB        DSET    FC=20,DEV=KB

CB1         BLK
            {
RDLA        BIN     X'0080'
            {
TB1         BLK
            {
TB2         BLK
            {
            PDIV

            ENTRY GO
```

3.2.5.2 Common workblocks

These are used to hold information required by more than one task, for example the current date and time, and may be used for passing information from one task to another; one task may write to a data item in a common workblock, and another task may subsequently access that data item. A common workblock (CWB) definition can be present in one or more terminal classes, and all the tasks in the terminal classes containing that common workblock declaration are able to access the information held in that common workblock.
For example:-

```
TERM    A0
TWB     TB1
CWB     CX2
CWB     CX1
TERM    B0
TWB     TB1
CWB     CX2
```

Each task in terminal class A0 will have a separate copy of terminal work block TB1 and will be allowed access to common work blocks CX1 and CX2. Tasks in terminal class B0 will have a separate copy of TB1 but they are only allowed access to common work block CX2. Information can be stored in a common work block by one task and accessed by another task. For example the supervisor may enter the current date as part of the start of day routine, then whenever a task requires the date it will obtain it from the date field in the common work block. Note that in terminal class A0 the order of work blocks is:-

```
TERM    A0
TWB     TB1
CWB     CX2
CWB     CX1
```

If CX1 and CX2 had been reversed then an error would occur when accessing work blocks in terminal class B0, as in terminal class B0 common work block CX2 is located after TB1.

IDENTIFIER	PAGE IN M04
CWB	1.3.16

3.2.5.3 User workblocks

In this context "user" is the work station operator. It may be necessary for the application program to maintain accumulators for each work station user, for example. However there need not be a fixed one-to-one relationship between work positions and users. There need not be the same number of users as work stations and they need not be assigned to particular stations.

To maintain user information, areas of memory are required which are associated with individual users and not with work stations. These areas of memory are called user work blocks (UWB). One or more user work block types may be defined for each terminal class.

Tasks may only refer to a user work block if it has been defined for their terminal class and then only when the program has executed a 'USE' instruction specifying the block identifier and index identifier, of that user work block. An index identifier is an integer in the range 1 to 999 which is used to differentiate between user work blocks of the same format.

For example:-

```
            TERM    A0
            TWB     TB1
            CWB     CX2
            UWB     UB1

UB1         BLK
CASID       STRG    6C'*'
DEP         BCD     12D'0'
WHDL        BCD     12D'0'
```

The number of copies of a user work block is entered in the configuration data, see M04 page 3.4.3 for details. If the user work block UB1 had been configured with four copies, then to access copy three of the user work block UB1 the following section of code would be executed.

```
            MOVE    INDX,=W'3'
            USE     UB1,INDX
            ADD     DEP,CSH
```

IDENTIFIER	PAGE IN M04
UWB	1.3.28

3.2.5.4 Dummy work blocks

Dummy work blocks can be used to redefine the data items forming another work block. For example:- it is only possible to read from disk into a string data item, so it is useful to have a work block which contains just the string data item and a dummy work block containing the field definitions to be imposed on this record. In the example below workblock TB1 contains the string data item which the record will be read into and DB1 contains the redefinition.

```
            TWB       TB1
            DWB       DB1(TB1)        Note work block to be redefined is TB1
.
TB1         BLK
BUF         STRG      66

DB1         DBLK                      Note dummy block definition begins DBLK
NAME        STRG      20
ADDR        STRG      30
POSTC       STRG      8
TELNO       STRG      8
```

For example if the contents of BUF are:-

```
FREDERIC SMYTHE      15, THE LOGWALK NEWTOWN      LL5 I11 789-1276
```

Then the contents of the data item identifiers forming the dummy work block DB1 will be as shown below.

```
NAME        FREDERIC SMYTHE

ADDR        15, THE LOGWALK NEWTOWN

POSTC       LL5 I11

TELNO       789-1276
```

IDENTIFIER	PAGE IN M04
DWB	1.3.20

3.2.5.5 Swappable workblocks

These can form the transition between work blocks held in the program and disk held files. Ordinary workblocks can have preset or empty values; each time the application is started other than via IPL, then the program held workblocks are set to the initial state. However swappable (disk held) workblocks contain what ever they held when the machine was last closed down.

The name "swappable work block" is derived from the fact that they can be "swapped" or exchanged between memory and disk storage, enabling values to be updated and held for future use. For a task to use a swapppable work block it must issue the USE command, and when it no longer requires the swappable work block the UNUSE command is issued which releases the block and copies it back to disk. It is not possible for two tasks to have the same swappable workblock in memory at the same point in time, though there can be more than one copy of a work block on disk and different tasks could access these different copies.

The USE and UNUSE commands have two arguments:- the block identifier (SB1) and a data item identifier. The block identifier specifies the name of the swappable work block e.g. SB1 and the data item identifier is used to specify which copy is required.

AT IPL THE THE DISK HELD FILE WILL BE RE-SET TO THE INITIAL CONTENTS.

```
            TERM    A0
            TWB     TB1
            SWB     SB1     Swappable work block
SB1         BLK             Start of definition for SB1
.................
Definitions for SB1
.................
```

The disk file for holding swappable work blocks is called $SWAP and is created at sytem load time, and the configuration file contains the details of the block definitions, see M04 page 3.4.4. Each copy of the work block is referenced by a data item identifier for example:-

```
            TERM    A0
            TWB     TB1
            CWB     CX2
            SWB     SB1
SB1         BLK
CASID       STRG    6C'*'
DEP         BCD     12D'0'
WHDL        BCD     12D'0'
```

If, for example, the swappable work block SB1 had four copies specified in the configuration file, then the following instructions would be required to access information held in the third copy of the swappable work block.

```
            MOVE    INDX,=W'3'
            USE     SB1,INDX
            ADD     DEP,CSH
```

IDENTIFIER	PAGE IN M04
SWB	1.3.25

DATA SET DECLARATION

```
            IDENT    MAIN
            DDIV
            TERM     T0
            TWB      TB1
            TWB      TB2
            CWB      CB1
            START    TGØ
DSKB        DSET     FC=20, DEV=KB
DSCAS       DSET     FC=12, DEV=TC, BUFL=100
DSSOPI      DSET     FC=10, DEV=S1
                ⎫
                ⎪
                ⎭
            PDIV
                ⎫
                ⎪
                ⎭
            END
```

3.2.6 Data set directive (DSET)

In a CREDIT program input and output devices are specified by terminal class, and they are defined using the dataset directive (DSET). This must occur after the appropriate terminal class directive (TERM) in the data division. The DSET directive is used to associate a data set identifier used in the procedure division with the TOSS device type and file code. With the DSET directive it is also possible to specify the buffer length to be used, and if the buffer is to be shared with any other device. The format of the DSET directive is:-

```
data-set-identifier DSET FC=file code
                         {,BUFL=decimal number  }
                         {,DEV=device type      }
                         {,BUFDS=buffer data set}
```

FC folllowed by a hexadecimal number gives the TOSS file code, in the example below the numeric display is using file code 41.

The keyword BUFL is followed by the length in decimal notation of a fixed length buffer to be used with this device. This must not be specified if the I/O operations on the device use a system buffer.
Device type (DEV) is an optional field and its sole purpose is as an aide-memoire to the programmer. It is recommended to use the TOSS device types as listed in M04, as at system generation these device codes will be used to assign the file codes to the required devices for each terminal class.

The keyword BUFDS specifies that the buffer is to be shared with another data set in the same terminal class.

For example:-

```
SCRN      DSET      FC=50,DEV=DY,BUFL=240
AUX       DSET      FC=41,DEV=DN,BUFDS=SCRN
```

Here the VDU (TOSS device type DY) shares a buffer with the numeric display (TOSS device type DN).

DIRECTIVE	PAGE IN M04
DSET	1.3.18

3.2.7 Data items

Before studying all the different types of data items available in CREDIT it is important to have a basic understanding of how information is held within a computer.

The system used by PTS to hold information is called binary; the presence of an item being denoted by the value one and the absence of an item by the value zero. An analogy can be made with a lightbulb, which is either lit (having information) or out (no information).

Within the computer this binary information is held in BITS - one bit holding one binary item; for convenience bits are assembled into larger units called words - each word consisting of sixteen bits. However as much of the work of a modern computer is handling character strings, and all possible characters can be represented by eight bits then the computer word has been divided into two equal sections called BYTES, these bytes are also subdivided into two hexadecimal (base sixteen) digits. This arrangement of units is shown in the table below:-

NAME	INT. REPRESENTATION	CONTENTS
BIT	|.|	can hold a zero or a one
DIGIT	|....|	four bits, can hold one hexadecimal character
BYTE	|........|	eight bits, can hold one ISO-7 character or two hexadecimal digits
WORD	|................|	sixteen bits, two characters, four hexadecimal digits

3.2.7.1 CREDIT data items

CREDIT has several types of data items, and use either bits, digits, bytes or words depending on how the data item is defined. The different types of data items used in CREDIT are described on the next few pages, and listed below.

BOOLEAN
BINARY
BINARY ARRAYS
BINARY CODED DECIMAL
BINARY CODED DECIMAL ARRAYS
STRING
STRING ARRAYS
LITERALS

3.2.7.2 Boolean data items (BOOL)

Each work block can have up to sixteen boolean data items and each boolean data item occupies one BIT. The bit is set to one if the data item holds the value TRUE and zero if it holds the value FALSE. The format of the boolean data item declaration is:-

data-item-identifier BOOL [value]

The data item identifier is the means by which this data item will be referenced in the PDIV.

The value is an optional field which allows the data-set-identifier to have a preset value. If the value is omitted then the default value of false is assumed. Valid values are shown below:-

TRUE
T
FALSE
F

Boolean data items must always be the first entries in the work block.

Below are some examples of boolean declarations.

NAME		VALUE	MEMORY CONTENTS
FLAG	BOOL	TRUE	1
NEW	BOOL	T	1
CHNGE	BOOL	FALSE	0
LITE	BOOL	F	0
DLTE	BOOL		0

IDENTIFIER	PAGE IN MO4
BOOL	1.3.15

BINARY

BIN	W'20'	⟶ X'0014'
BIN	X 'F3'	⟶ X'00F3'
BIN	5X' 315FA'	⟶ X'15FA'
BIN	4D'0100'	⟶ X'0100'
BIN	D'123456'	⟶ X'3456'
BIN	1D'5'	⟶ X'0005'
BIN	C'NO'	⟶ X'4E4F'
BIN	C'ANO'	⟶ X'4E4F'
BIN		⟶ X'0000'

3.2.7.3 Binary data items (BIN)

A binary data item occupies one word (sixteen bits) and can be used for holding items shown in the table below. The interpretation given to the contents of the data item is dependant on the value code, the default value being type word (W) and value zero. The binary data item declaration format is shown below:

data-item-identifier BIN [[[length]value-type]['value']]

The data item identifier is the means by which this data item will be referenced in the PDIV.

The value type is one of the following:-

Type	Int. representation	Notes
W	One word (1*16 bits)	Number in range -32768 to +32767
C	Two bytes (2*8 bits)	Can contain two ISO-7 characters
X	Four hexadecimal digits (4*4 bits)	Hexadecimal number
D	Three digit decimal number with sign (4*4 bits)	Unused elements set to zero, the sign bit is B for positive, D for negative

Below are listed examples of binary data items, some with values assigned, together with the internal representation.

Data item identifier		Value	Machine form hexadecimal
SP1	BIN	W'32767'	7FFF
SP2	BIN	4D'100'	B100
SP3	BIN	X'FF'	00FF
SP4	BIN	2C'NO'	4E4F
SP5	BIN		0000
SP6	BIN	C'DOMINO'	4E4F
SP7	BIN	3D	B000

Note:-

In SP2 the specified length includes the sign.
In SP6 only the last two characters are held in the item.
In SP7 the largest possible value is 999 although it was only specified as a two digit field plus sign.

IDENTIFIER	PAGE IN M04
BIN	1.3.12

DATA-ITEM-SPECIFICATION

	LENGTH ITEM SIZE	VALUE TYPE	'VALUE'
BCD	NUMBER OF (4 BITS) DIGITS	D	'DECIMAL NUMBER'
		X	'HEXADECIMAL INTEGER'
BIN	1 WORD	W	'DECIMAL NUMBER'
	NUMBER OF (4 BITS) DIGITS	D	'DECIMAL NUMBER'
	NUMBER OF (8 BITS) CHARACTERS	C	'CHARACTER STRING'
	NUMBER OF (4 BITS) HEXADECIMAL DIGITS	X	'HEXADECIMAL INTEGER'
STRG	NUMBER OF (8 BITS) CHARACTERS	C	'CHARACTER STRING'
	NUMBER OF (4 BITS) HEXADECIMAL DIGITS	X	'HEXADECIMAL INTEGER'

3.2.7.4 Binary coded decimal data items (BCD)

BCD data items consist of a number of digits (4 bits) holding either a decimal (base 10) or hexadecimal (base 16) number. The maximum number of digits that can be held in one data item is 255, though in the case of decimal numbers the first digit position is reserved for the sign. The format of this declaration is:-

```
data-item-identifier  BCD  { length, value type [ 'value' ]  }
                           { [length,] value type 'value'    }
                           { [length[, value type]] 'value'  }
```

The data item identifier is the means by which this data item will be referenced in the PDIV.

Length is the number of digits to be used for this data item, including the sign.

Value type - this specifies whether the data item is to be hexadecimal digits (value type X) or decimal digits (value type D), the default type is 'D'

The value field allows a preset value to be given to a data item, as shown in the table below.

Contents of value field	Contents of data item	Example	
		Specified	Contents
Not Given	Zero		0
Less Than specified Length	Value right justified within data item	6D'23'	B'0023
Greater than specified length	Least significant digits will be held	4D'2600'	B'00

Note:-
Either the value or the type and length must be specified, the value must be enclosed in quotes.

Examples of BCD data declarations are given below:-

Data item identifier		Value	Machine form hexadecimal	
BCD1	BCD	6D'-23'	D00023	negative value sign set to D
BCD2	BCD	3D'100'	B100	rounded up to evan no. of bytes
BCD3	BCD	X'FF'	FF	
BCD4	BCD	6D	000000	
BCD5	BCD	'12300'	B12300	(implied D)
BCD6	BCD	X'ACE560'	ACE560	
BCD7	BCD	5X	000000	

DATA-ITEM-SPECIFICATION

	LENGTH ITEM SIZE	VALUE TYPE	'VALUE'
BCD	NUMBER OF (4 BITS) DIGITS	D	'DECIMAL NUMBER'
		X	'HEXADECIMAL INTEGER'
BIN	1 WORD	W	'DECIMAL NUMBER'
	NUMBER OF (4 BITS) DIGITS	D	'DECIMAL NUMBER'
	NUMBER OF (8 BITS) CHARACTERS	C	'CHARACTER STRING'
	NUMBER OF (4 BITS) HEXADECIMAL DIGITS	X	'HEXADECIMAL INTEGER'
STRG	NUMBER OF (8 BITS) CHARACTERS	C	'CHARACTER STRING'
	NUMBER OF (4 BITS) HEXADECIMAL DIGITS	X	'HEXADECIMAL INTEGER'

3.2.7.5 String data items (STRG)

String data items are composed of 1-4095 bytes, each byte holding one alpha-numeric character. The format of the string data item identifier is:-

```
data-item-identifier STRG {[[ Length,] Value type]  'Value'}
                          { [ Length [,Value type]] 'Value'}
                          { Length, Value type ['Value']}
```

The data item identifier is the means by which this data item will be referenced in the PDIV.

Length - is the number of characters that will make up the string

Value type - is either character (type C) or hexadecimal (type X), the default being 'C'.

Value - is an optional field and allows a data item to have a preset value

Contents of value field	Contents of data item	Example	
		Specified	Contents
Not given	Spaces		
Less than specified length	Value left justified within data item. Last character repeated	6C'AC'	ACCCCC
		6C'AC '	AC
Greater than specified length	Leftmost characters will be stored	4C'BRANCH'	BRAN

Data item identifier	Type	Value	Machine form hexadecimal
STRG1	STRG	6C	202020202020
STRG2	STRG	3C'ABC'	414243
STRG3	STRG	X'FF'	FF
STRG4	STRG	6C'ABC'	414243434343
STRG5	STRG	6C'ABC '	414243202020
STRG6	STRG	5C'BANK ID'	42414E4820
STRG7	STRG	5X	00000

IDENTIFIER	PAGE IN M04
STRG	1.3.23

3.2.7.6 Arrays

There are three types of arrays in CREDIT: BCDI, BINI and STRGI. They can be either one or two dimensional. The subscript must be a binary data item. The maximum subscript is 32767 for one-dimensional arrays. For two-dimensional arrays neither subscript can exceed 255. Arrays occupy two entries in a work-block, unless an array is the last item in the workblock when it occupies only one. The rules for the storage of the values are the same as for BCD, BIN and STRG data items described above.

The general format for an array declaration is:-

data-item-identifier type (S1[,S2])[,[length] type] ['value'...]

S1 - is the "row" subscript

S2- is the "column" subscript for two dimensional arrays

'value'... - this enables the array to be initialised. If fewer values are provided than there are elements in the array, then all remaining elements are filled with the last provided value, e.g.

```
BRANCH    STRGI     (40),10C'LONDON ','ROME ','PARIS ','*'
```

This sets up a forty element array containing branch locations; as only three exist at this point the unused elements are filled with asterisks. The first six elements of this array are shown below:-

element	contents
(1)	LONDON
(2)	ROME
(3)	PARIS
(4)	**********
(5)	**********
(6)	**********

If the contents of an element are less then the string length then the last character is repeated, if for example the declaration had taken the following form:-

```
BRANCH    STRGI     (40),10C'LONDON','ROME','PARIS','*'
```

Then the contents of the first six elements would be:-

element	contents
(1)	LONDONNNNN
(2)	ROMEEEEEE
(3)	PARISSSSS
(4)	**********
(5)	**********
(6)	**********

For a two dimension array the data items should be ordered by rows then columns, for example:-

```
MONQUA    STRGI    (3,4),10C'JANUARY ','FEBRUARY ','MARCH ','APRIL ',
                   'MAY','JUNE ','JULY ','AUGUST ','SEPTEMBER ',
                   'OCTOBER ','NOVEMBER ','DECEMBER '
```

This would set up a table of the months in each of the four quarters of the year, the month in the quarter being the first subscript, the second the quarter of the year. The machine representation would be:-

Element	Contents	Element	Contents	Element	Contents
(1,1)	JANUARY	(2,1)	FEBRUARY	(3,1)	MARCH
(1,2)	APRIL	(2,2)	MAY	(3,2)	JUNE
(1,3)	JULY	(2,3)	AUGUST	(3,3)	SEPTEMBER
(1,4)	OCTOBER	(2,4)	NOVEMBER	(3,4)	DECEMBER

The binary declaration is similar to that for strings, an example of a two dimensional binary array is given below.

```
TAB    BINI    (4,4),'1','2','3','4','5','6','7','8','9','10',X'B',
               X'C',X'D',X'E',X'F',X'10'
```

Element	Contents	Element	Contents	Element	Contents	Element	Contents
(1,1)	0001	(2,1)	0002	(3,1)	0003	(4,1)	0004
(1,2)	0005	(2,2)	0006	(3,2)	0007	(4,2)	0008
(1,3)	0009	(2,3)	000A	(3,3)	000B	(4,3)	000C
(1,4)	000D	(2,4)	000E	(3,4)	000F	(4,4)	0010

IDENTIFIER	PAGE IN M04
BCDI	1.3.11
BINI	1.3.13
STRGI	1.3.24

3.2.8 Literals - overview

With CREDIT there are four distinct catagories of literals; these are literal constants, keytables, pictures and format lists; after translation each of these literals will be held in separate pools.

3.2.8.1 Literal constants

These are the normal form of constants used in a program and have the general form:-

=[Value type]'value'

Value type is one of those listed in the table below

Type	Int. representation	Notes
W	One word (1*16 bits)	Number in range -32768 to +32767
C	(n) bytes (n*8 bits)	Can contain (n) ISO-7 characters
X	(n) hexadecimal digits (n*4 bits)	Hexadecimal number n digits long
D	(n-1) digit decimal number with sign (n*4 bits)	Unused elements set to hex F Sign bit is B for positive, D for negative

Note:-
Literal constants can never form the destination part of an instruction, in that a constant may be added or moved to a data item, but a data item can not be added or moved to a literal constant.

Examples of literals are:-

='2'	Typeless literal
=W'45'	One word containing value 45
=C'BANK ID.'	Character string
=6D'-97892'	BCD constant
=X'2030'	Hexadecimal constant

3.2.8.2 Keytables

These are used for holding lists of character codes which could be used to terminate keyboard input. Only hexadecimal (type X) or character (type C) data items may be used in a keytable. A description of keytables is given in section 6.2.3. The format of a keytable declaration is:-

```
    key-table-name KTAB {literal constants  }
                        {EQU data items     }
```

For example

```
    BSP         EQU         X'05'
    CLEAR       EQU         X'19'
    CLR2        EQU         X'00'
    EOI         EQU         X'12'
    CANC1       EQU         X'14'
    CANC2       EQU         X'15'
    SPKTAB1     KTAB        BSP,CLEAR,CLR2,EOI,CANC1,CANC2
```

3.2.8.3 Picture literals

These are used when formatting numeric items for display or printing purposes; either for output of for re-displaying an input item. Picture literals can only be used with the FMEL instruction. Some examples of picture literals are shown in the table below:

Picture	Data item	Result
'AAA999'	BFF0456	0456
'XXY-XX'	2702	27-02
'XXY-XX'	FFFF	
'9909'	B123	1203
'F+**V9'	DF11	*-1.1
'99,99'	B123	12,34
'99B99'	6521	65 02

3.2.8.4 Format lists

These are used to hold format layouts for the output of information. Examples of format lists are shown below, and are described in more detail in section 6.5.

```
ERFM01  FRMT
        FSL
        FTEXT     'TOO FEW INPUT CHARACTERS'
        FMEND
*
ERFM02  FRMT
        FCOPY     =X'2031'
        FTEXT     'RETYPE ANWER YES OR NO: '
        FMEND
```

4. INSTRUCTIONS

Instructions direct the input, processing and output of information. They specify the actions to be carried out by the computer, and direct the sequence of events.

The general form of CREDIT instructions is:-

```
[STATEMENT-IDENTIFIER]   INSTRUCTION-MNEMONIC   OPERAND-1[,OPERAND-2...]
```

STATEMENT IDENTIFIER - this identifies a point within the program and is used in branch and entry instructions; a statement identifier need not be on the same line as the statement, e.g.

```
FRED1     ADD       A,B       this statement is identified as FRED1
```

could also be written on two lines, e.g.

```
FRED1
          ADD       A,B
```

INSTRUCTION MNEMONIC - this specifies the basic operation to be performed by the instruction. There are nine groups of instructions, and the instruction mnemonic must be derived from one of these groups; see appendix A for a list of instructions and categories. In the above example the INSTRUCTION MNEMONIC was ADD, one of the arithmetic group of instructions.

OPERANDS - these contain the operational part of the instructions, their significance being different for each instruction. Each instruction described here and in M04 refers to operands from left to right as OPERAND-1, OPERAND-2 etc.

The groups of instructions available in CREDIT are as follows:

Arithmetic
Branch
Input/Output
Logical
Scheduling
Storage Control
String
Subroutine control
Format I/O control

GROUPS OF INSTRUCTIONS

- ARITHMETIC INSTRUCTIONS
- LOGICAL INSTRUCTIONS
- STRING INSTRUCTIONS
- BRANCH INSTRUCTIONS
- SUBROUTINE INSTRUCTIONS
- INPUT/OUTPUT INSTRUCTIONS
- SCHEDULING INSTRUCTIONS
- STORAGE CONTROL INSTRUCTION

4.1 Arithmetic instructions

These have two operands, and consist of the following instruction mnemonics:-

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

The arithmetic instructions ADD, SUB, MUL, DIV and DVR operate on either BCD or BIN data items or arrays. Both operands must be of the same type, e.g. both BIN or both BCD.

The CMP instruction operates on BCD, BIN or STRG data items or arrays. Both operands must be of the same type.

The MOVE instruction can be used with BIN, BCD or STRG data items or arrays, and the operands need not be of the same type.

After the execution of most CREDIT instructions the status will be held in a special register by the Interpreter, called the Condition Register (CR). This can be used to determine the path the program takes on different conditions.

Arithmetic instructions will cause the Condition Register to be set as shown in the table below.

Value	Meaning
0	Zero result
1	Positive result
2	Negative result
3	Arithmetic overflow occurred

ADD INSTRUCTION

```
                        IDENT     MAIN
                        DDIV
                        TERM      T0
                        TWB       TB1
                        START     TAGO

              TB1       BLK
10            CATOT     BOOL      FALSE
11            EOTAPE    BOOL      FALSE
10            INDEX     BIN       'O'
11            INLEN     BIN       'O'
12            LAMP1     BIN       X'0200'
13            LAMP9     BIN       X'0100'
14            SUBACC    BCD       8
15            TWORK1    BCD       8
16            ARRAY     BCD1      (5),10D'O'
19            DMSTR     STRG      1C' '

                        PDIV
                        ENTRY     TAGO

              TAGO
02 14 15                ADD       SUBACC, TWORK1

02 16 10 15             ADD       ARRAY (INDEX), TWORK1
```

4.1.1 The Add instruction (ADD)

The ADD instruction adds operand-2 to operand-1 and stores the result in operand-1. Execution of this instruction will affect the Condition Register, as shown in the table in section 4.1.

Examples of the ADD instruction

```
        ADD         ACC,DEP
```

This increases the contents of the data item ACC by the contents of the data item DEP.

```
        ADD         LOOP,=W'1'
```

This increases the contents of the binary data item LOOP by one

```
        ADD         CASH(TID,USER),DEP
```

This increases the element in array CASH referenced by the subscripts TID and USER by the amount held in the data item DEP.

4.1.2 The Subtract instruction (SUB)

The SUB instruction subtracts the contents of operand-2 from the contents of operand-1; execution of this instruction will alter the Condition Register as shown on page 4.1.1.

Examples of the SUB instruction

```
        SUB         LOOP,=W'1'
```

If LOOP had an initial value of 10 then after execution of this statement it would contain 9.

```
        SUB         TEST,=W'-1'
```

If test held an initial value of 5 then after this statement had been executed it would hold the value 6.

INSTRUCTION	PAGE IN M04
ADD	I.4.24
SUB	I.4.149

4.1.3 The Divide instruction (DIV)

The DIV instruction divides the contents of operand-1 by the contents of operand-2 and stores the result in operand-1, any remainder being ignored. Division by zero results in overflow, thus giving a value of 3 in the CR. The Condition Register contents are described on page 4.1.1.

Example of the DIV instruction

```
DIV     COM,=W'100'
```

This divides the data item COM by 100, if COM had an initial value of 378, then after this instruction COM would have the value 3.

```
COM/100          = 3.78
Rounded down     = 3

Result in COM    = 3
```

4.1.4 The Divide Rounded instruction (DVR)

The DVR instruction divides the contents of operand-1 by the contents of operand-2; 0.5 is then added to the result and the rounded down result stored in operand-1. The Condition Register is set as described on page 4.1.1

Example of the DVR instruction

```
DVR     COM,=W'100'
```

If the data item COM had the initial value 378 then after the division the result would be 4, as shown below.

```
COM/100        = 3.78
+ 0.5          = 4.28
Rounded down   = 4

Result in COM = 4
```

4.1.5 The Multiply instruction (MUL)

The MUL instruction multiplies the contents of operand-1 by the contents of operand-2 and stores the result in operand-1. The CR is set as described on page 4.1.1.

Example of the MUL instruction

```
MUL     AMM,XRATE
```

The data item AMM is multiplied by the data item XRATE. If the initial value of AMM was 1700 and XRATE 450 then after this statement had been executed AMM would contain the value 765000. If the receiving data item is not large enough to hold the result, its contents will be undefined and the CR will be set to to overflow (value 3).

INSTRUCTION	PAGE IN M04
DIV	1.4.68
DVR	1.4.91
MUL	1.4.129

4.1.6 The Compare instruction (CMP)

The CMP instruction compares two data items of the same type for similarity. It sets the condition register to one of the values shown below, according to the relationship between the two data items.

When the two data items are of different lengths, the comparison will be executed as follows:

- For string data items the shortest data item will be extended (by the Interpreter) with blank characters (X'20') from the right.
- For decimal data items the shortest data item will be extended (by the Interpreter) with zero digits (X'0').

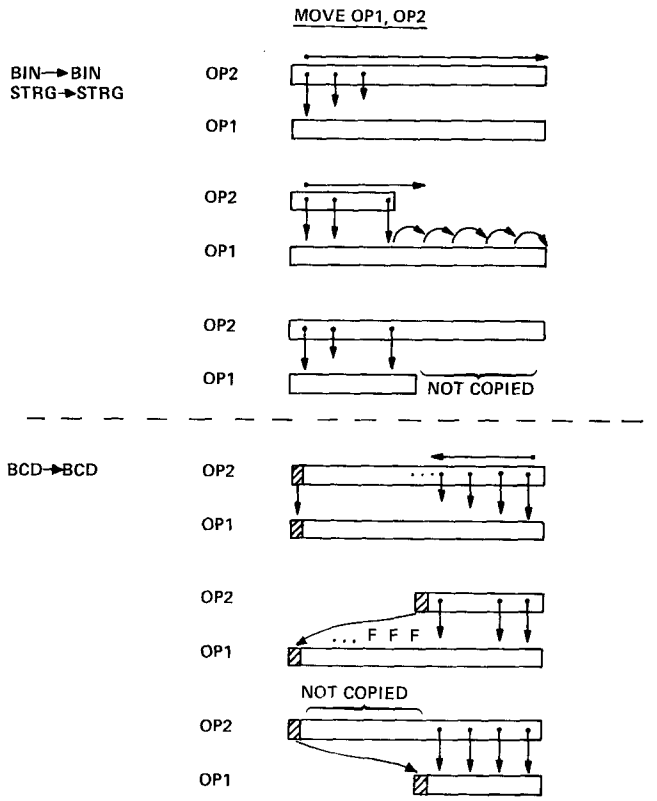
The values held in the Condition Register after execution of this instruction are:-

VALUE	MEANING
0	Operand-1 = Operand-2
1	Operand-1 > Operand-2
2	Operand-1 < Operand-2

Examples of the CMP instruction

```
         CMP      SPBINW1,SPBINW2          Two items of like type

         CMP      SPBINW3,=W`97'           Comparison with a constant
         CMP      SPINPUT,=C`YES'

CON1     EQU      W`97'
         CMP      SPBINW3,CON1             Use of an EQU constant
```

INSTRUCTION	PAGE IN MO4
CMP	1.4.62


MOVE OP1, OP2
BIN→BIN
STRG→STRG
OP2
OP1
OP2
OP1
OP2
OP1
NOT COPIED
BCD→BCD
OP2
OP1
OP2
OP1
F F F
NOT COPIED
OP2
OP1

4.1.7 The Move instruction (MOVE)

The MOVE instruction moves the contents of operand-2 to operand-1. The operands can be of type BIN, BCD or STRG, though transfer from BIN to STRG or STRG to BIN is not permitted.

The MOVE can be used for transferring numeric information only between the data types linked by arrows in the diagram below.

BIN <---> BIN <---> BCD <---> BCD <---> STRG <---> STRG

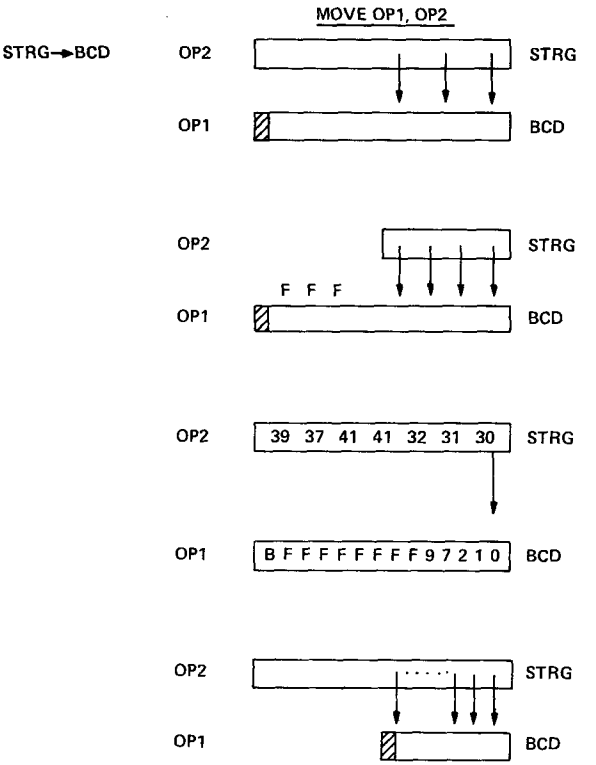
The rules for moving of data items is given on pages 1.4.127-128 of M04. In summary, moving to a shorter data item causes truncation of information, and moving to a longer data item results in padding.

Examples of the MOVE instruction

```
MOVE      OUT,=C'PLEASE ENTER USER CODE'
```

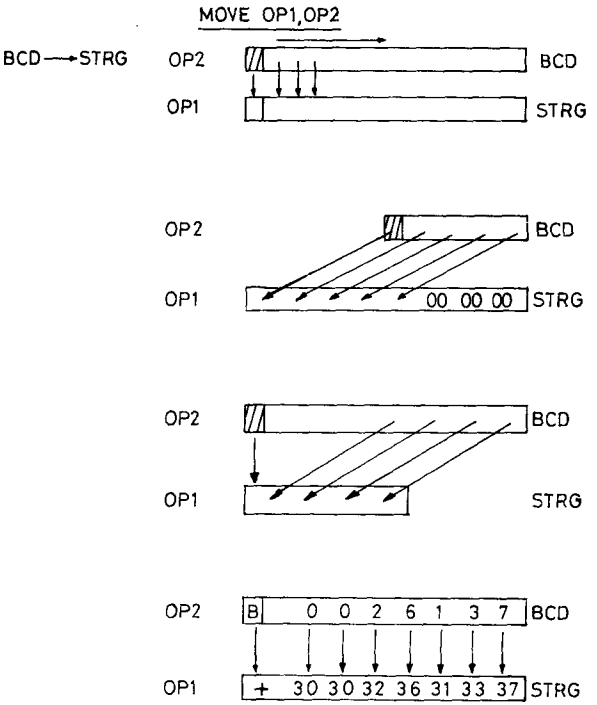
If OUT has been defined as STRG with a length of twenty-two characters then the character string would be transferred to OUT. If OUT were longer than twenty two characters then the last character in the string would be repeated. For example, if OUT had been defined as length twenty-five, then after this instruction had been executed it would contain "PLEASE ENTER USER CODEEEE". If OUT had been defined as length ten, then it would contain "PLEASE ENT", only the left hand ten characters being transferred.

INSTRUCTION	PAGE IN M04
MOVE	1.4.127


MOVE OP1, OP2
STRG→BCD
OP2
STRG
OP1
BCD
OP2
STRG
F F F
OP1
BCD
OP2
39 37 41 41 32 31 30
STRG
OP1
B F F F F F F F F 9 7 2 1 0
BCD
OP2
STRG
OP1
BCD

```
MOVE      OUTX,=C'PLEASE ENTER YEAR E.G. 1979 : '
```

If, in the above example, OUTX had been defined as BCD then it would contain '1979' as non-numeric characters are not transferred from STRG to BCD. Note that after execution the contents of OUTX will have been right justified, have the sign digit set and all unused digits will be set to X'F' (X'F' is the null digit for BCD items). If a transfer of a number to either a BCD or BIN data item is requested, and the number is too large to be held by that data item then its contents will be uncertain and the Condition Register will be set to 3 (Overflow).


MOVE OP1,OP2
BCD ⟶ STRG
OP2
OP1
BCD
STRG
OP2
OP1
BCD
00 00 00
STRG
OP2
OP1
BCD
STRG
OP2
OP1
B
0 0 2 6 1 3 7
BCD
+ 30 30 32 36 31 33 37
STRG

```
MOVE        FLDA,NUMB
```

If FLDA has been declared as a STRG data item, four bytes long, and NUMB is a BCD data item containing the value +123456, then the result in FLDA will be

```
+456
```

since moving from BCD to STRG always results in the sign being moved first. The remaining digits are moved, and converted, from left to right, but in this case the receiving field is too short, hence only the rightmost digits are transferred.

4.2 Logical instructions

These are single operand instructions and allow logical operations on boolean data items. Boolean data items are used for holding such things as status flags. Note that the Condition Register will be set to the previous contents of the data item after execution of a logical instruction.

The available instructions are

CLEAR	Clear a boolean data item (result binary zero)
INV	Invert a data item (reverse its state)
SET	Set a data item (result binary one)
TEST	Compare with zero (false) and set condition register

Examples of logical instructions

```
CLEAR     FLAG
```

This sets the boolean data item FLAG to FALSE (zero)

```
INV       FLAG
```

The state of FLAG is reversed - if it was FALSE it will now be TRUE

INSTRUCTION	PAGE IN M04
CLEAR	1.4.61
INV	1.4.117
SET	1.4.146
TEST	1.4.155

4.3 String instructions

These instructions are for the manipulation of character strings.

The available instructions are

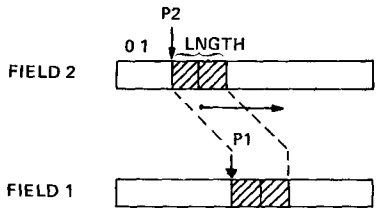
COPY
MATCH
INSERT
DELETE
XCOPY

These instructions operate on string data items, with two exceptions
. The XCOPY command can be used with both STRG and BCD data items.
. The COPY command can be used with BIN, BCD and STRG data items.

The string handling commands have two different types of operand.
. Character strings (or BIN and BCD data items in the case of the two exceptions listed above).
. Pointers and lengths, held in binary data items.

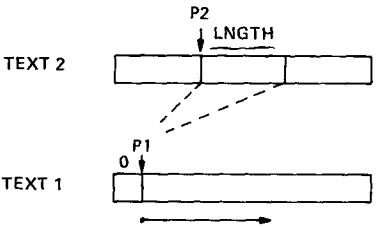
COPY INSTRUCTION

COPY FIELD1,P1,LNGTH,FIELD2,P2



INSERT INSTRUCTION

INSRT TEXT1, P1, LNGTH, TEXT2, P2



ORIGINAL CONTENTS SHIFTED TO THE RIGHT

* COND.REG.=3 IF NON-SPACE
OR NON-ZERO
CHARACTER SHIFTED OUT

4.3.1 The Copy instruction (COPY)

The COPY instruction is used to move a number of decimal digits or bytes from one data item to another of the same type. Both data items must be STRG or both must be BCD or both must be BIN.

The instruction format is:-

```
COPY    Dest, Start, No., Source, Start-2
```

Dest - is the data item which is to have information copied into it.

Start - is a binary data item containing a pointer to the position in Dest where the copied information is to begin.

No. - is a binary data item containing the number of characters (bytes) or decimal digits (half bytes) to be copied from Source to Dest.

Source - is the data item, part or all of which will be copied into Dest.

Start-2 - is a binary data item containing a pointer to the start of the information in Source that is to be copied into Dest.

The pointers (Start and Start-2) assume that the first byte location is zero, so to access the second byte or digit the pointer must have a value of one.

Example of the COPY instruction

```
MOVE    S1,=W'0'
MOVE    S2,=W'4'
MOVE    S3,=W'1'
COPY    DEST,S1,S2,SRC,S3
```

If SRC had been defined as a STRG data item ten bytes long, containing the string "XCURRENCY", and DEST as a STRG data item four bytes long, then after execution DEST would contain "CURR".

If SRC had been defined as a BCD data item ten digits long, containing the decimal number "523012350", and DEST as a BCD data item four digits long, then after execution DEST would contain "2301". Note that the sign position in a BCD data item can be changed by the program, by use of this instruction. The Condition Register is not affected by the execution of this instruction.

INSTRUCTION	PAGE IN M04
COPY	1.4.63

4.3.2 The Extended Copy instruction (XCOPY)

The XCOPY instruction moves bytes between any non-boolean data items. It always copies at byte level, regardless of data types.

The instruction format is:-

```
XCOPY    Dest, Start, No., Source, Start-2
```

Dest - is the data item which is to have information copied into it.

Start - is a binary data item containing a pointer to the position in Dest where the copied information is to begin.

No. - is a binary data item containing the number of characters to be copied from Source to Dest.

Source - is the data item, part or all of which will be copied into Dest.

Start-2 - is a binary data item containing a pointer to the start of the information in source that is to be copied into dest.

The pointers (Start and Start-2) assume that the first byte location is zero, so to access the second byte or digit the pointer must have a value of one.

Example of the XCOPY instruction

```
MOVE     S1,=W'0'
MOVE     S2,=W'4'
MOVE     S3,=W'1'
XCOPY    DEST,S1,S2,SRC,S3
```

SRC has been defined as a STRG data item ten bytes long and contains the string 'ABCDEFGHI', and DEST as a BCD data item eight digits long. Then after execution of this statement DEST will contain '42434445', the hexadecimal equivalent of the character string 'BCDE'.

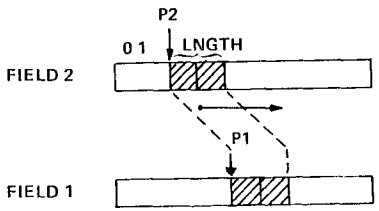
If SRC had been defined as a BCD data item fourteen digits long and contained the hexadecimal characters '24435552525553', and DEST as a STRG data item four characters long, then after execution of this instruction DEST would contain 'CURR'.

The Condition Register is not affected by the execution of this instruction.

INSTRUCTION	PAGE IN M04
XCOPY	1.4.173

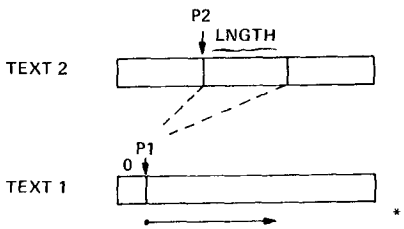
COPY INSTRUCTION

COPY FIELD1,P1,LNGTH,FIELD2,P2



INSERT INSTRUCTION

INSRT TEXT1, P1, LNGTH, TEXT2, P2



ORIGINAL CONTENTS SHIFTED TO THE RIGHT

* COND.REG.=3 IF NON-SPACE
OR NON-ZERO
CHARACTER SHIFTED OUT

4.3.3 The Insert instruction (INSRT)

The INSRT instruction is used to insert a character string into an existing character string, the existing contents being shifted to the right to produce the required space.

The instruction format is:-

```
INSRT    Dest, Start, No., Source, Start-2
```

Dest - is the data item which is to have information inserted into it.

Start - is a binary data item containing a pointer to the position in Dest where the inserted information is to commence.

No. - is a binary data item containing the number of characters from Source to be inserted into Dest.

Source - is the data item, part or all of which will be inserted into Dest

Start-2 - is a binary data item containing a pointer to the start of the information in Source that is to be inserted into Dest.

The pointers (Start and Start-2) assume that the first byte location is zero, so to access the second byte or digit the pointer must have a value of one.

If a non-space or non-zero character is shifted out of Dest, the Condition Register will be set to 3 (overflow). Each character shifted out of the dataitem is lost.

Example of the INSRT instruction

```
MOVE     S1,=W'5'
MOVE     S2,=W'4'
MOVE     S3,=W'4'
INSRT    DEST,S1,S2,SRC,S3
```

If the initial contents of the string data item DEST was 'CODE:=N/A ' and the contents of SRC was '23456789' then after this operation DEST will contain 'CODE:=6789'. Note that the previous contents have been shifted to the right and as the field length was only ten characters the last four are lost. One data item could be saved by writing the instruction in the form shown below:-

```
INSRT    DEST,S1,S2,SRC,S2
```

If the initial contents of DEST nad been 'ABCDEFGHIJKL' and SRC contained '23456789' then after execution of this statement DEST would contain 'ABCDE6789FGHI'.

The details of this operation are shown below:

INSTRUCTION	PAGE IN M04
INSRT	1.4.116

DEST

ABCDEFGHIJKLM SRC

23456789

Move four spaces into DEST

ABCDE FGHI take four characters from SRC starting at at character position four (the first character position is zero)

6789

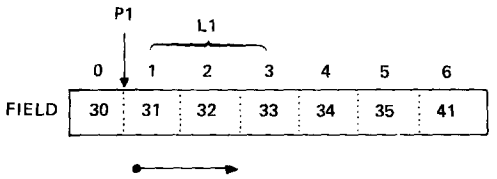
Now put the two parts together

ABCDE6789FGHI

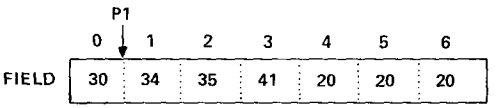
If a non-blank or non-zero character is lost off the end of the receiving field then the Condition Register will be set to 3 (overflow).

DELETE INSTRUCTION

DLETE FIELD, P1, L1



RESULT AFTER EXECUTION



4.3.4 The Delete instruction (DLETE)

The DLETE instruction is used to remove characters from a STRG data item, the characters remaining to the right of the deletion are then shifted to the left to fill the gap caused by the deleted characters, and spaces are used to fill from the right.

The Condition Register is not affected by the execution of this instruction.

The instruction format is:-

```
DLETE     String, Start, No.
```

String - is the character string which contains the item to be deleted

Start - this is a binary data item giving the character position for the deletion to begin

No. - this is a binary data item and contains the number of characters to be deleted

The pointer (Start) assumes that the first byte location is zero, so to access the second byte the pointer must have a value of one.

Example of the DLETE instruction

```
MOVE      S1,=W'6'
MOVE      S2,=W'4'
DLETE     DEST,S1,S2
```

If the initial contents of DEST was 'SMITH MRS PAT', after the above section of program had been executed DEST would contain 'SMITH PAT '.

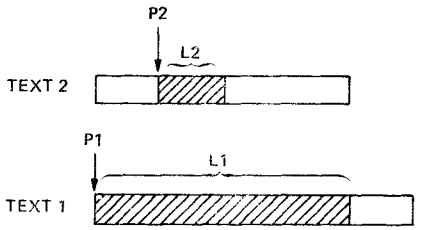
The details of this operation are shown below

```
Delete characters            'SMITH ....PAT'      (. is deleted char)

Move remaining characters    'SMITH PAT....'

Fill from right with spaces  'SMITH PAT    '
```

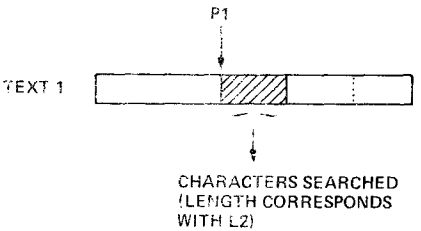
INSTRUCTION	PAGE IN M04
DLETE	1.4.69

MATCH INSTRUCTION

MATCH TEXT1,P1,L1,TEXT 2,P2,L2



IF MATCH OCCURS THEN COND.REG.=0
RESULT:



4.3.5 The Match instruction (MATCH)

The MATCH instruction is used to search for the occurence of a string or partof a string within another string. The condition register will be set to zero if a match is found, or 4 if there is no match.

If a match occurs then the second operand (Start-1) will be set to the position where the match was found; if there was no match then the contents will be undefined.

The instruction format is.-

MATCH String-1, Start-1, No.-1, String-2, Start-2, No.-2

String-1 - is the character string to be searched

Start-1 - is the position at which the search will start in String-1.

No.-1 - is the number of characters to be searched in String-1.

String-2 - is the data item containing the string to be matched.

Start-2 - is a binary data item giving the position in String-2 for the start of the string that is to be compared with String-1.

No.-2 - is a binary data item containing the number of characters in String-2 that are to be matched with String-1. This data item must contain a number that is less than or equal to No.-1.

The two pointers (Start-1 and Start-2) assume that the first character position is zero.

Example of the MATCH instruction

```
MOVE    S1,=W'0'
MOVE    S2,=W'27'
MOVE    S3,=W'3'
MATCH   VAL,S1,S2,INP,S3,S3
BE      OK
```

If VAL contained '001,002,003,004,005,006,007' and INP 'ID=005', then after the MATCH instruction has been executed control will be transferred to the statement identifier OK and S1 will be set to '16'.

INSTRUCTION	PAGE IN MO4
MATCH	1.4.125

4.4 Branch instructions

An important consideration when writing any real time application, is that all potential error situations are detected and handled correctly; to aid the programmer achieve this objective many CREDIT commands use the Condition Register to record the status after execution.

CREDIT provides a wide variety of branch instructions to transfer control according to the contents of the condition register. There are also branch instructions which allow the comparison of two data items and branch if a certain condition occurs, to branch on boolean data items, indexed branches and unconditional branches.

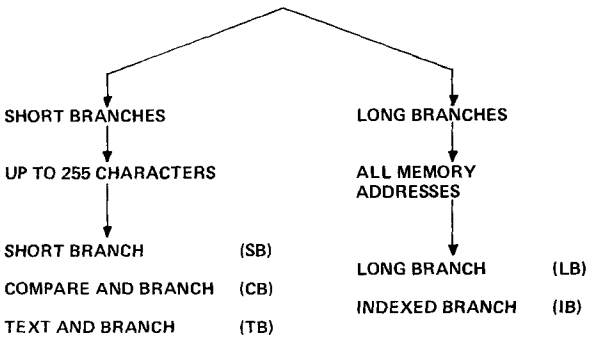
The CREDIT Translator produces two kinds of branches:

. Short branches where the destination is within 255 bytes of the branch

. Long branches for all other situations.

Short branches have a one byte displacement for holding the destination address; however a long branch has an index to T:BAT (the long branch table) where the address of the destination is held.

BRANCH INSTRUCTIONS



4.4.1 Unconditional branches

The unconditional branch instruction enables the transfer of control to the statement label identifier specified in the operand.

The instruction format is:-

```
B    OP1
```

OP1 is the statement label identifier to which the program will branch after having encountered this instruction.

Example of an unconditional branch

```
B    FRED1
```

The program will always branch to the statement label identifier FRED1 when this statement is encountered.

INSTRUCTION	PAGE IN M04
B	1.4.25

BRANCH WITH CONDITION MASK

```
            LB          1,          CONT1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
GR          EQU         1
            LB          GR, CONT1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
GR          EQU         1
            CB          GR, INLEN, CBINO, RDERR2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            IB          INDEX, SYS20, SYS40
```

4.4.2 Branch on condition mask

These instructions enable a branch to be made according to the contents of a condition mask.

The general format of this instruction is:-

```
{SB}
{ B}   [<COND>,]<statement identifier>
{LB}
```

If the branch instruction 'B' is used, the Translator decides whether it is a long or short branch and produces the appropriate object code. SB and LB are the mnemonics for long and short branch respectivly.

The <COND> is optional, and if present gives the appropriate condition mask for the branch, as shown in the table below; if this field is ommitted then it becomes an unconditional branch.

Cond. Code	Cause of this condition
0	Zero result from arithmetic operation Equality found with Compare instruction Logical data item had previous value of false I/O operation completed satisfactorly
1	Positive result from arithmetic operation Operand-1 greater than Operand-2 in Compare instruction End of file detected on I/O operation
2	Negative result from arithmetic operation Operand-1 less than Operand-2 in Compare instruction I/O error on an I/O operation
3	Arithmetic overflow Beginning or End of device on I/O operation
4	Inverse of condition code zero
5	Inverse of condition code one
6	Inverse of condition code two
7	Unconditional branch

Example of conditional branch

```
        B     5,L1
```

This will cause a branch to statement identifier L1 if the condition mask is equal to five (negative, or operand one less than or equal to operand two, or no end of file encountered by I/O operation)

INSTRUCTION	PAGE IN M04
B	1.4.25

4.4.3 Mnemonic branches

To make the branch instructions easier to use the code can be replaced by a mnemonic branch. The mnemonic branches use the Condition Register to establish whether or not control is to be transferred, and can be divided into three sections:

. Those for use after I/O operations

. Those for use after the CMP instruction

. Those for use after arithmetic instructions

These instructions have only one operand and this contains the statement label identifier to which control will be transferred, should the condition be satisfied.

4.4.3.1 Conditional branch after I/O instructions

Both input and output instructions cause the Condition Register to be set when they are executed. It is regarded as good program design to include checks to detect any errors that have occurred as a result of an input or output operation, and have routines to handle these situations. If an error has occurred more detail can be obtained with the XSTAT instruction, as shown in section 6.1.1.

The branch commands available for use after input or output operations are shown in the table below.

Note:

Not all these conditions can be generated by all I/O operations, for example, EOF condition will not occur when writing to a display.

The following mnemonics are used for these branches:

BEOF	Branch if End of File
BERR	Branch if Error
BEOD	Branch if End of Device
BNOK	Branch if not OK (Same as BERR)
BNEOF	Branch if not End of File
BNERR	Branch if no Error
BOK	Branch if OK (Same as BNERR)

INSTRUCTION	PAGE IN M04
BEOF	1.4.28
BERR	1.4.29
BEOD	1.4.26
BNOK	1.4.39
BNEOF	1.4.34
BNERR	1.4.35
BOK	1.4.43

4.4.3.2 Conditional branch after compare

Two operands may be compared using the CMP command described in section 4.1.6; as a result of this comparison the Condition Register is set, and can be used by branch instructions.

These branch instructions have one operand which is the statement identifier to which control will be passed if the condition defined in the branch mnemonic matches that set by the compare.

The example below shows the branch mnemonics that can be used after the compare instruction.

Example of conditional branching after compare

```
CMP     OLD,NEW          COMPARE TWO FIELDS
BE      L1               BRANCH IF EQUAL
BG      L2               OLD > NEW
B       L3               OLD < NEW
```

If OLD contained the value 5 and NEW the value 5 then control would be passed to the statement at label L1. If OLD contained the value 6 and NEW the value 5 then control would be passed to the statement at label L2, otherwise control is passed to L3 unconditionally since OLD must be less than NEW.

The conditional branches may be one of the following:

```
BE    Branch if equal     (Operand-1 = Operand-2)
BG    Branch if greater   (Operand-1 > Operand-2)
BL    Branch if less      (Operand-1 < Operand-2)
BNE   Branch if not equal    (Operand-1 ¬= Operand-2)
BNG   Branch if not greater  (Operand-1 ¬> Operand-2)
BNL   Branch if not less     (Operand-1 ¬< Operand-2)
```

INSTRUCTION	PAGE IN M04
BE	1.4.27
BG	1.4.30
BL	1.4.31
BNE	1.4.33
BNG	1.4.36
BNL	1.4.37

4.4.3.3 Conditional branch after arithmetic instruction

After an arithmetic instruction has been obeyed the Condition Register will contain information about the resultant value, if it was positive, zero or negative or if overflow had occurred. The branch instructions which can be used after arithmetic instructions are:-

BN	Branch if result < 0
BNN	Branch if result > 0 or = 0
BNP	Branch if result < 0 or = 0
BNZ	Branch if result < 0 or > 0
BOFL	Branch if overflow occurred
BP	Branch if result > 0
BZ	Branch if result = 0

Example of conditional branch

The following section is from a program which is being used for processing binary (BIN) data items.

```
        MUL     AMM,XRATE           CONVERTED AMOUNT
        BOFL    OVF57               OVERFLOW
        BNP     MC17                NO POSITIVE AMOUNT
        MOVE    WK1,AMM             WORKING STORE VAR.
        MUL     WK1,=W'175'
        BOFL    OVF58               OVERFLOW
        DVR     WK1,=W'100'         AMM*1.75
        MOVE    BAL,WK1             STORE
        B       MC18
MC17    MOVE    BAL,AMM
MC18                                RESULT NOW IN BAL
```

If AMM contained 1700 and XRATE 450, then after the multiply command had been executed, the contents of AMM would be undefined, though the overflow bit in the condition register would have been set. When the branch on overflow (BOFL) command is encountered a branch would be made to the statement identifier OVF57.

If AMM contained 0 and XRATL 450, then a branch would be made to the statement identifier MC17.

Following the multiplication of WK1 by the constant 175 overflow may occur (the largest binary number that the machine can store is 32767), and a test is made. If overflow has occurred then a branch would be made to OVF58. However, as there is no risk of overflow or division by zero at the DVR command, no checks follow that instruction.

INSTRUCTION	PAGE IN M04
BN	1.4.32
BNN	1.4.38
BNP	1.4.40
BNZ	1.4.41
BOFL	1.4.42
BP	1.4.44
BZ	1.4.45

4.4.4 Compare and branch instructions

These instructions are combined compare and branch instructions; but they can only be used where a short branch would be generated, hence the destination address of the branch must be within 255 bytes of the current address. If the destination address is greater then 255 bytes from the current address then the following section of code could be used.

```
CMP     A,B
BE      L1
```

However for short branch situations the following combined instruction is used.

```
CBE     A,B,L1
```

The rules for data types in the compare instruction also apply to the data types in the compare and branch instructions. This is a three operand instruction of the format:-

CB<type> data item 1, data item 2, statement identifier

The first operand must be a data item identifier.

The second operand is either a data item identifier or a literal.

The third operand is the statement identifier which will be branched to if the condition specified in the compare and branch instruction is satisfied by the first two operands.

The type is one of E,G,L,NE,NG,NL and is used to form the mnemonics listed below.

Examples of the compare and branch instruction

```
CBL     BAL,AMM,OVD         Branch to OVD if BAL is less than AMM
CBE     AMM,MAXL,SPC1       Branch to SPC1 if AMM is equal to MAXL
CBNE    LOOP,=W'1',ROUND    Branch to ROUND if LOOP is not equal to 1
CBL     RATE,IRR,GO         Branch to GO if RATE is less than IRR
CBG     AMT,FIX,ERR         Branch to ERR if AMT is greater than FIX
CBNG    ACC,MIN,READ        Branch to READ if ACC is not greater than MIN
```

INSTRUCTION	PAGE IN M04
CBE	1.4.49
CBG	1.4.51
CBL	1.4.53
CBNE	1.4.55
CBNG	1.4.57
CBNL	1.4.59

4.4.5 Test and branch

These instructions can only be used for branching dependent on the condition of boolean data items, and have the following format:-

```
command        boolean variable, statement identifier
```

The command can either be test and branch if true (TBT) or, test and branch if false (TBF).

The first operand specifies the boolean data item on which the decision to branch or not will be made.

The second operand is the statement identifier to which control will be passed should operand one satisfy the criteria of the command.

Example of the test and branch command

```
TBT        STAT,MC17
```

This will cause a branch to statement label MC17, if STAT holds the value TRUE otherwise the next consecutive instruction will be obeyed.

INSTRUCTION	PAGE IN MO4
TBF	1.4.52
TBT	1.4.53

4.4.6 Indexed branch instructions

The indexed branch command (IB) is used to generate a long or short branch to one of a number of statement labels depending upon the value of an index.

The format of the indexed branch command is :-

```
IB        Index, Label-1, Label-2,..... Label-n
```

Index - is a binary data item containing the index to be used by the branch.

Label-1 etc. - these are a list of statement identifiers to which the program may branch depending upon the value in the index. If the index held the value one then this would cause a branch to the statement at Label-1, if the index held the value 2 then a branch would be made to the statement at Label-2 and so on.

If the index contains the value zero, or a value greater than the number of statement label supplied, then the next consecutive instruction will be executed.

Example of an indexed branch

```
IB        SPBINW2,READIN,DUMMEY,KEOI,KTFWD,KTBWD,KTHOME,                C
          KTLDOWN,KTLEFT,KTRIGHT,KTUP,KENTER
SUB       SPBINW2,=W'14'
```

SPBINW2 is the binary index used for controlling this indexed branch; where the program branches to depends on the contents of this data item.

Contents of SPBINW2	Statement identifier to which control will be passed
1	READIN
2	DUMMEY
3	KEOI
4	KTFWD
5	KTBWD
6	KTHOME
7	KTLDOWN
8	KTLEFT
9	KTRIGHT
10	KTDOWN
11	KTUP
12	KENTER

If SPBINW2 contains the value zero, or a value greater then twelve then the next consecutive statement will be executed, in this case the subtract command.

INSTRUCTION	PAGE IN M04
IB	1.4.114

5. SUBROUTINE HANDLING

5.1 Introduction

Subroutines are usually small sections of a program for performing a single function, be it initialising data items, displaying items on a visual display screen, or carrying out a modulus eleven check, for example.

Writing programs as a series of subroutines can reduce development, testing and maintenance time compared with a 'monolithic' approach. As smaller units are easier to understand, testing can be carried out on each subroutine in turn.

If a CREDIT subroutine is located in a different module to the routine that is to call it, then the calling module must contain an external directive (EXT) giving the subroutine name, and the module containing the subroutine itself must contain an entry directive (ENTRY) to match.

Example:

```
Main module                    Subroutine module

EXT  SUB1                      SUB1     PROC

PERF SUB1
```

Subroutines in CREDIT are enclosed in directives starting with the procedure directive (PROC) and ending with the procedure end directive (PEND).

Example:

```
XCH      PROC
          <
          >        CREDIT statements forming
          <        the subroutine called XCH
         PEND
```

The subroutine name is located in the label field of the PROC directive; in the above example the subroutine name is XCH, and this is the name that must appear in the ENTRY and EXT directives, if performed from another module.

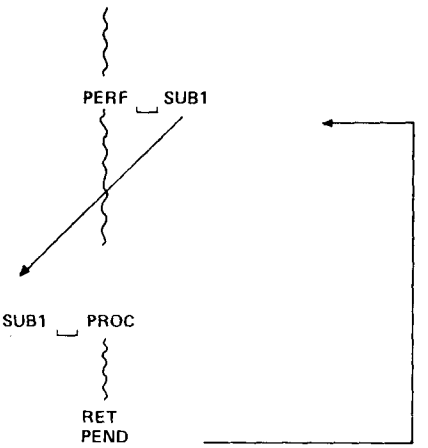
5.2. Execution of a subroutine

The transfer of control to the subroutine is achieved using the perform command (PERF) or the perform indexed command (PERFI), and the return of control to the calling routine with the return command (RET).

The PERF and PERFI command both store on the system stack the address of the next instruction to be obeyed after a normal return from the subroutine, along with other information on the task module. Each stack entry occupies six bytes, and the stack currently has a default size of 128 bytes though this may be altered by using the stack directive. If, for example, a large number of embedded performs are to be made the size may have to be increased, and if no embedded calls are to be made the stack size may be reduced.

REFERENCE	PAGE IN M04
ENTRY	1.2.7
EXT	1.2.9
PERF	1.4.134
PERFI	1.4.135
PEND	1.2.17
PROC	1.2.21
RET	1.4.139

CREDIT SUBROUTINE IN THE SAME MODULE



The perform command has the following format:-

```
PERF    subname{,p1{...,pn}}
```

where - subname is the name of the subroutine to be executed.
- p1 to pn are actual parameters which will be passed to the subroutine.

The indexed perform command has the following format:-

```
PERFI   index,s1{,s2...,sn}
```

where - index is a binary index to be used for selecting the subroutine that is to be executed
- s1,s2...,sn is a list of subroutine names, the first entry being regarded as entry one, so if the variable being used for the index has a value 2 then the 2nd subroutine will be activated.

The return command has the following format:-

```
RET     <opt. byte dis.>
```

The optional byte displacement specifies the number of bytes which are to be added to the return address before the return command is executed.

The RET instruction transfers control back to the calling routine at the instruction after the PERF command or PLIST directive (see below), and must therefore be the last logical instruction in a subroutine.

The address to which the return will be made is held on the system stack; it is the byte displacement from the perform instruction. The perform instruction may vary in length depending on the number of parameters and the type of addressing adopted; when the RET instruction is encountered control is returned to that instruction. It is possible to add a displacement to the return command to enable a return to a subsequent instruction, as described above.

Example:

```
PERF    ABC,A,B
B       L1
ADD     A,B
RET
```

On encountering the above return instruction, control will be returned to the branch instruction, which will transfer control to the instruction at statement identifier L1. However as the branch instruction occupies two bytes; if the subroutine ABC is terminated by the instruction:-

```
RET     2
```

then control will be returned to the ADD instruction, as the branch instruction occupies two bytes.

REFERENCE	PAGE IN M04
PERF	1.4.134
PERFI	1.4.135
RET	1.4.139

5.2 Parameter handling

5.2.1 General rules

A subroutine can access directly any data item defined in the data division which is available to the calling routine. In addition a subroutine may have formal parameters, which are local names listed in the operand section of the PROC directive and used solely within that subroutine. When the subroutine is executed then the actual parameters will be substituted for these formal parameters. The actual parameters are variable names used within the calling routine. The actual parameters are specified in the PERF command after the subroutine name, and if the PERFI command is used then they are specified in the PLIST directive immediately after the PERFI command. There can be up to eight formal parameters in a subroutine, depending on the contents of the LITADR option in the directive OPTNS.

The parameters in the calling program are called the "actual parameters" and those in the subroutine the "formal parameters". The valid types for actual parameters are listed below.

boolean (BOOL) data items
binary (BIN) data items
binary arrays (BINI)
binary coded decimal data items (BCD)
binary coded decimal arrays (BCDI)
string data items (STRG)
string arrays (STRGI)
data set identifiers
literals, but not those with an unspecified type or of type X

The indexed perform may pass parameters to a subroutine, in which case all the subroutines in the subroutine list will require the same number of parameters. The parameters to be transferred are specified in the parameter list directive (PLIST) located immediately after the PERFI command. The format of the PLIST directive is :-

```
        PLIST    p1{,...pn}
```

p1{,...pn} is the list of actual parameters which will be passed accross to the subroutine.

Example:

```
        PERFI    SUB1,SUB2,SUB3,SUB4
        PLIST    ACCNO,NAME,=D'1',BINW4
```

Note:-

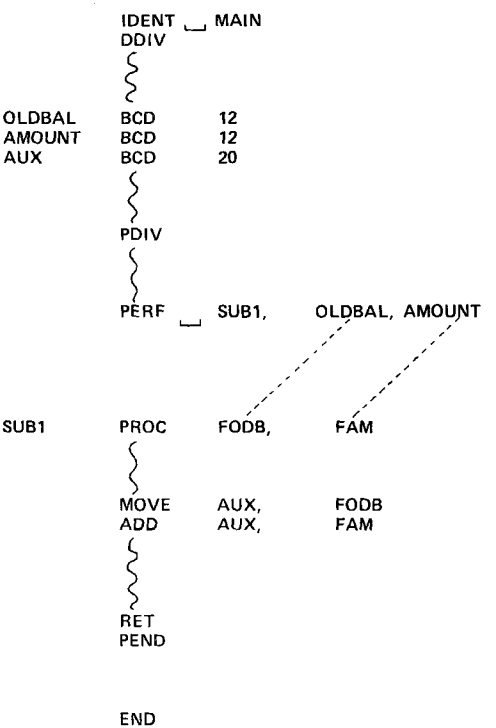
If an array element is being passed to a subroutine, then the array and the subscript must be passed as separate parameters. In addition, the formal parameter for the array itself must be followed by open and close parenthesis, to indicate that it is an array.

Example:

```
        PERF     SUB2,ARRAY,INDEX,AMOUNT

SUB2    PROC     FARR(),FINDX,FAM
```

DIRECTIVE	PAGE IN M04
PLIST	1.4.210

ACTUAL/FORMAL PARAMETERS



```
          SUBROUTINE  (FORMAL PARAMETERS, ARE
                       KEY TABLE, FORMAT LIST, LITERAL)
          PERF        SUB1, LENGTH, FORM1, KTAB1, =D'1'
SUB1      PROC        INLEN, $FORM, $KTB, $LIT
          ADD         TRANR, $LIT
          K1          DSKB, INBUF, $KTB, INLEN, INDEX
          EDWRT       DSVOU, $FORM
          PEND
```

5.2.2 Literals, keytables and format lists as parameters.

There are two ways of passing literals, keytables and format lists to subroutines, as follows:

Method 1

The actual parameter is specified in the normal way, and the formal parameter is given a name which starts with a $ (dollar sign). This tells the translator that the actual parameter to be substituted at execution time is one of the three types defined above, e.g:

```
        PERF    SUB1,LENGTH,FORM1,KTAB1,=D'1'

SUB1    PROC    INLEN,$FRM,$KTB,$LIT1
         |
        ADD     INLEN,$LIT1
         |
        RET
        PEND
```

Method 2

The formal parameter is not given a name starting with a dollar sign. In this case, the directives shown below must appear immediately after the PROC statement at the start of the subroutine.

PFRMT when using format lists
PLIT when using literals
PKTAB when using keytables

These are always required when ADRMOD is set to 2 in the OPTNS directive, and the formal parameter must not then be preceded by a $ sign, e.g:

```
        PERF  SUB1,LENGTH,FORM1,KTAB1,=D'1'

SUB1    PROC  INLEN,FRM,KTB,LIT
        PFRMT FRM
        PKTAB KTB
        PLIT  LIT
         |
        ADD   INLEN,LIT
         |
        RET
        PEND
```

A summary of rules for passing format lists, keytables and literals is shown on the next page.

DIRECTIVE	PAGE IN M04
PFRMT	1.2.18
PKTAB	1.2.19
PLIT	1.2.20

Summary of rule for passing literals, keytables and format lists

name	PROC	FORM1	(ADRMOD=2)	$ not required
	<opt>	FORM1		
name	PROC	$FORM1	(ADRMOD=1)	$ required
name	PROC	FORM1	(ADRMOD=1)	$ not required
	<opt>	FORM1		

name is the name of the subroutine
<opt> is either PFRMT, PLIT or PKTAB.

7. TASK SCHEDULING AND ACTIVATION

7.1 Dispatcher queue

It has been previously stated that, in a system when more than one task exists, the tasks are 'considered for dispatching' when a LKM is performed. The mechanism for task dispatching and queuing, of tasks will now be outlined.

The scheduling of tasks is performed by the TOSS Monitor: at system start, all the tasks are placed in the Dispatcher queue, and the first task is then activated, i.e. starts execution.

A task runs in User mode, which means that when an I/O instruction is executed, control passes to the Monitor. The Monitor then starts the I/O operation. The task can not continue until the I/O is complete, assuming Wait is used, and this means that the task is put at the back of the dispatcher queue, which operates on a FIFO principle, and another task may get control. This principle is necessary particularly for keyboard input, which is relatively slow, since otherwise the keyboard input task would hold up the other tasks in the system. It is quite possible that the dispatcher queue can be empty, if all tasks are waiting for I/O to be completed. In this case the Monitor is in an 'idle loop', until such time as I/O completes for one of the tasks. There are other ways in which the dispatcher queue can be affected and these are the instructions:

PAUSE inhibits execution of the task, until restarted by another task; during this time the task is pending, and is not considered for dispatching.

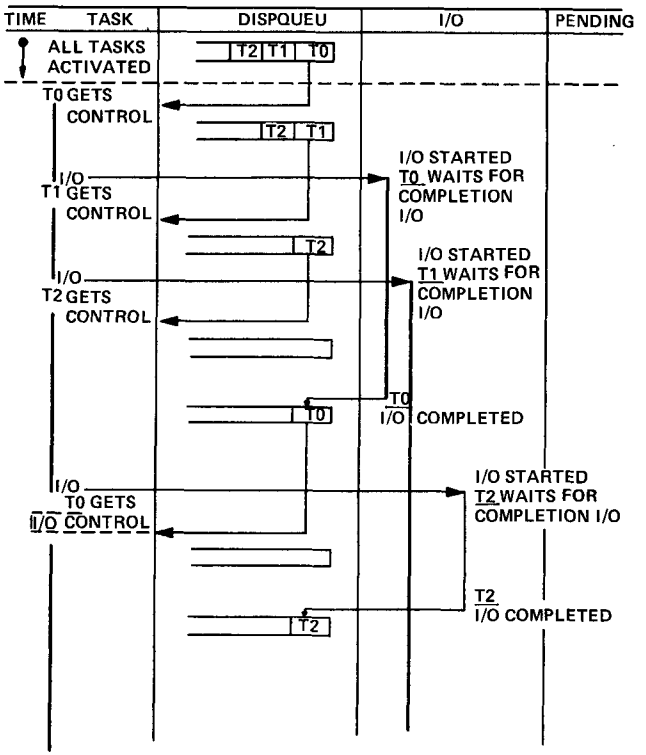
RSTRT restarts the specified task; the task does not necessarily get control, but is placed at the back of the dispatcher queue.

EXIT terminates the task, and it ceases to exist; task tables and all references to the task are deleted.

ACTV activates a task, which may or may not have already been active and performed an EXIT. Activation may be at any statement that contains a label. Again, the task may not get control, but is placed in the dispatcher queue.

Keyword	Page in manual
PAUSE	MO4 1.4.133
RSTRT	MO4 1.4.142
EXIT	MO4 1.4.106
ACTV	MO4 1.4.20

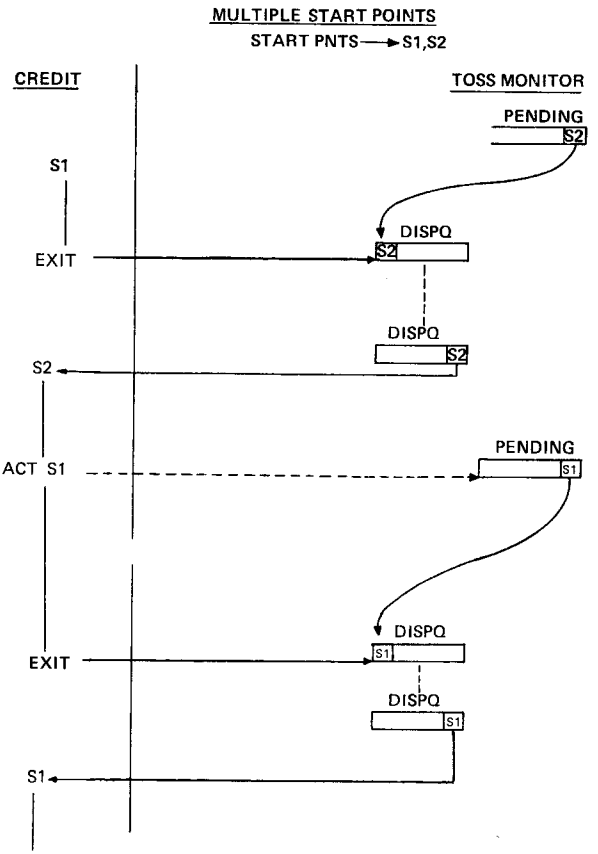
SCHEDULING OF TASKS



7.2 More than one start point

It is possible to specify more than one start point in a module. At system start, the task will be activated at both start points. However, since the task can not exist twice, one start point is placed in the dispatcher queue, and the other in the pending queue.

When the task EXITs, the task is deactivated, and the second start point is removed from the pending queue and placed in the dispatcher queue, and is thereby activated at the second start point.


MULTIPLE START POINTS
START PNTS → S1,S2
CREDIT
TOSS MONITOR
PENDING
S2
S1
DISPQ
EXIT
S2
DISPQ
S2
S2
PENDING
ACT S1
S1
DISPQ
EXIT
S1
DISPQ
S1
S1

8. INTERTASK COMMUNICATION

8.1 Introduction

It is possible for tasks to communicate with each other within the system, by issuing 'messages'. These messages may be directed to a specific task or may be general, i.e. issued to all or any of the other tasks. Similarily, tasks may request a message from a specific task, or from any task that has issued a message.

Two queues are maintained in the system for unaddressed message requests, one for input (read) messages, and one for output (write) messages. Only one queue may have entries at one point in time. When both queues have an entry the message transfer takes place and both items are removed from the queue.

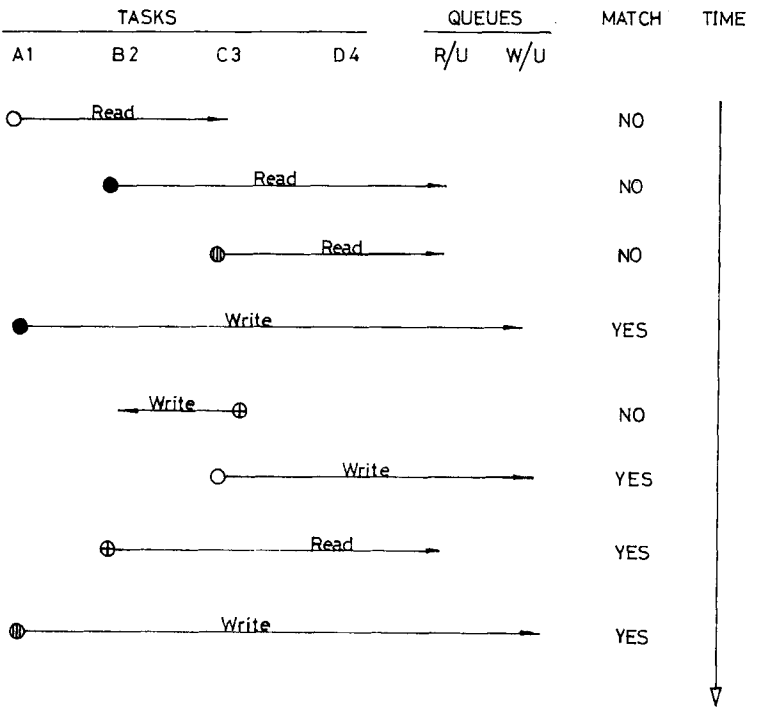
This queueing sytem works on the first in first out (FIFO) principal, with one exception that will be seen later in this section. The simplest case is as follows:-

Task A issues an unaddressed write command. This request is placed in the unaddressed write queue, until such a time as Task B issues a read; at this point the two requests are satisfied, and the queues cleared. Task A would then be able to resume, assuming the instruction had the wait bit set.

In addition, a task may issue a specific read for a message from another task, and this may result in the reception of a message specifically directed to the reading task, or one from the queue of unaddressed writes, providing the message in the latter case was issued by the task from which the current task is now requesting a message, and no matching addressed request exists.

Furthermore, a task may issue a specific write directed at another task, and in this case, if no read is oustanding from that task in either of the queues, this request is queued until such a time as the specified task issues a read directed at the task sending the message. The .NW and WAIT functions apply to intertask I/O in the same way as for conventional I/O.

INTERTASK COMMUNICATION



8.1.1 Unaddressed read and write

These use the standard read and write commands, the data set directive being the TOSS file code defined for intertask communication. Examples of these instructions are shown below:-

```
READ   DSIC,MESBUF,SIZE
WRITE  DSIC,MESBUF,SIZE
```

8.1.2 Addressed read and write

These use the same read and write commands as random I/O, and again the data set directive will be using the TOSS file code defined for intertask communication, and in place of the record number will be the appropriate task identifier. If an addressed read is issued, then the unaddressed queue will first be checked to see if the task specified has already issued an unaddressed write, if none is found then the addressed queue will be searched, and if a match was not found then the task will be suspended until the read can be satisfied. Examples of these instructions are shown below:-

```
RREAD   DSIC,MESBUF,SIZE,TASKID
RWRITE  DSIC,MESBUF,SIZE,TASKID
```

8.1.3 Examples of intertask communication

Four tasks exist in the system; A1, B2, C3 and D4.

Action	Result
A1 issues a READ ADDRESSED to C3	Request queued in C3
B2 issues a READ UNADDRESSED	Request in R/U queue
C3 issues a READ UNADDRESSED	Request in R/U queue
A1 issues a WRITE UNADDRESSED	Matched to 1st in R/U queue message passed to B2
C3 issues a WRITE ADDRESSED to B2	Request queued on B2
C3 issues a WRITE UNADDRESSED	Matched to A1 R/A on C3 message passed to A1
B2 issues a READ UNADDRESSED	Matched to W/A on C3 message passed to C3
A1 issues a WRITE UNADDRESSED	Matched to 1st in R/U queue message passed to C3

Keyword	Page in MU4
READ	1.4.137
RREAD	1.4.140
RWRITE	1.4.143
WRITE	1.4.169

9. SCREEN MANAGEMENT

9.1 Introduction

Screen management consists of several CREDIT subroutines utilizing format I/O control, format I/O control was described in section 6.6 of this manual. This package enables a complete screen of information to be displayed, typically with a number of areas where the operator has to enter or change items. These "input fields" can initially contain an existing value or they could be set to blank.

Each prompt and input field is in a format list in the application program. The format list is attached with the ATTFMT instruction prior to screen management being called, the DISPLAY being done within the package.

With display units that permit both high and low intensity display, the items entered from the keyboard will be displayed in high intensity mode; those items of text emanating from the format list will be displayed in low intensity.

This package is held as CREDIT source code in a file called SCREEN, in the user area SCREEN on the system pack. Before the package can be used it has to be copied to the application user area. As the package is written in CREDIT, the routines can be easily modified; for example, the package as written expects the DDIV of the application to be held on a file called SPDDIV; if this is not the case, then the DDUM statement in the screen management package must be altered accordingly.

9.2 Requirements of screen management

Screen management requires that a number of definitions are made within the application. These are three keytables, some extra data items, data sets for keyboard, display and associated printer, a format table of error messages and a number of checking routines. The data item, data sets and format control statements must be included in the DDIV for the application. It is usual to group all other items in a file called 'SPLITT', this file being incorporated into the screen management module at translation time; the screen management package use the INCLUDE directive to bring across the information from SPLITT.

If the module SPLITT is not used, then the INCLUDE directive in the module SCREEN must be removed, or if a module name other than SPLITT is used, the INCLUDE directive must be changed to the new name.

Note:

It is not possible to have an IDENT statement in files which are to be included via the INCLUDE directive. A sample layout of SPLITT is included at the end of this chapter.

9.2.1 Data items

The data items given in the table below are required for use by the screen management package and must be defined in the DDIV of the application program. These data items must be accessable to the task which is initiating the perform of screen management.

Data items required by screen management

Name	Type	Size	Use
SPBINW1	BINARY	One word	Work variable
SPBINW2	BINARY	One word	"
SPBINW3	BINARY	One word	"
SPBINW4	BINARY	One word	"
SPCHANGE	BOOLEAN	One bit	Fields changed
SPPROMPT	BOOLEAN	One bit	Display form
SPINPUT	STRING	Length greater then Largest input field	Input item
SPERCALL	BOOLEAN	One bit	Work variable
SPSTRGW1	STRING	Greater than one	"

9.2.2 Data sets

Screen management uses three data sets, a printer, display unit and a keyboard. These data sets must be defined in the application program with the names shown in the table below. The buffer for the display unit must be large enough to hold the longest line plus the associated control characters; the buffer for the print unit must be at least the same size as the display. The two buffers may be shared. The data division must include the FMTCTL directive to link the two devices and enable some of the format control I/O instructions to be used by screen management.

Data sets used by screen management

Name	Description
SPDSPRT	Printer data set
SPDSSCRN	Display data set
SPDSDYKB	keyboard data set

9.2.3 Entry points

There are a number of entry points for the package, as described below. The required entry points must be defined as external (EXT) in the application program. The entry points are subroutine names, hence they will be accessed by a perform (PERF) or indexed perform (PERFI) instruction in the application program.

SPCLRN The prompts and titles described in the attached format list will be displayed on the screen, only if the boolean data item SPPROMPT has the value 'TRUE'; if SPPROMT has the value 'FALSE' then they will not be displayed. Irrespective of the value of SPPROMPT, the old contents of the data items in the format list will be displayed, the cursor will be placed at the first data field and will wait for the user to enter information, or move the cursor to another data field.

SPCLRS The prompts and titles described in the attached format list will be displayed on the screen, only if the boolean data item SPPROMPT has the value 'TRUE'; if SPPROMT has the value 'FALSE' then they will not be displayed. Irrespective of the value of SPPROMPT, the old contents of the data items in the format list will be displayed only if the no clear (NCLR) option is specified in the FKI command, the other fields will be left blank; the cursor will be placed at the first data field and will wait for the user to enter information, or move the cursor to another data field.

SPCLRA The prompts and titles described in the attached format list will be displayed on the screen, only if the boolean data item SPPROMPT has the value 'TRUE'; if SPPROMT has the value 'FALSE' then they will not be displayed. Irrespective of the value of SPPROMPT the old contents of the data items in the format list will not be displayed, the cursor will be placed at the first data field and will wait for the user to enter information, or move the cursor to another data field. The use of this call with an SPPROMPT value of FALSE should be avoided as it results in a blank screen.

SPERR When screen management detects an error the acoustic alarm is sounded and an error message is displayed on the last line of the screen. This entry point allows errors detected in the application program to be displayed in a similar manner. After the error message has been displayed control will return to the application. The binary data item SPBINW4 is used as an index to the format table SPFTBERR which holds the format lists for the error messages.

SPERR2 Like SPERR this entry point allows an error detected in the application to be displayed; in addition it allows the errored field to be corrected before control is transferred back to the application. The binary data item SPBINW4 is used as an index to the format table SPFTBERR which contains the format lists for the error messages.

9.2.4 Keytables

Screen management uses three keytables, one to control editing of the current field, one to check the first character entered in a field and one to check all subsequent characters in the field. The hexadecimal values generated by the actual keys on the keyboard used in the application must be present in the keytables, and the entries are position dependant. As supplied the package expects the keytables to be called SPKTAB1, SPKTAB2 and SPKTAB3.

SPKTAB1 is used to check the first character entered in a field. If the character entered matched to an entry in the keytable then the appropriate action will be taken, for example if the 'TAB LEFT' key had been entered then the first input field on the current line is made current.

If the character entered did not match with any of the table entry and was not a valid alphanumeric character then the acoustic alarm in the display unit will be sounded, and the cursor remains at the start of the field.

If the character entered was a valid alphanumeric character, and did not match with the keytable then it will be transferred to the string data item SPINPUT and the remainder of the field on the display unit will be filled with periods '.'.

The field size is either the limit specified in the MAXL option on the DYKI instruction, or the length of the string, if the MAXL is not specified.

SPKTAB2 is used to check the second and subsequent characters entered. If a match is found with the keytable then the specified action will be taken.

If the character entered was not found in the keytable, and it was not an alphanumeric character then the acoustic alarm will be sounded and the cursor remains in its current position.

If a valid alphanumeric character was entered and a match was not found with the keytable then the character is placed in the string data item SPINPUT and the cursor is positioned at the next character; if the field has been filled then all characters entered until a valid terminator is entered, except when the NEOI flag is set in the DYKI instruction. In this case, when a field is filled the next field will become current, the cursor is then positioned at the first input position of that field.

SPKTAB3 is used for holding the character codes to be used for editing the current field. When the editing has been completed and the field contains the desired contents then one of the function keys will be used to make the next field current. If the DYKI instruction has the NEOI flag set then when the current field is filled, the next field will become current.

Note:

If a keytable entry is not required then the value of X'FF' should be entered in the approriate position. This is often called the NOKEY value.

If a 6236 keyboard is being used then it is necessary to include a special character conversion table CTAB01, which specifies the code to be generated by each key. When a table entry is not required it should be set to the bell character X'07'.

An example of the three keytables is given in SPLITT at the end of this section. The position and description of the function key codes within the three keytables is described in the next section.

9.2.5 Keytable entries

9.2.5.1 Editor functions

Non-destructive space - Position 1 in SPKTAB3

When this key is pressed the cursor is moved one position to the right; the acoustic alarm is sounded if an attempt is made to go beyond the current last character of the input field. Also it is not possible to position the cursor beyond the current contents of the field.

If for example, the field had been specified with a length of ten characters, and currently the field held only five characters then the acoustic alarm would be sounded if an attempt was made to move the cursor to position six.

If the NEOI flag is set in the DYKI instruction, and this key is pressed when the cursor is in the last position of the field a return will be from edit and the next field will become current.

If the NEOI flag was not set and the cursor was currently in the last position of the field when this key was pressed, the acoustic alarm will be sounded and the cursor position will not be changed.

Non-destructive backspace - Position 2 in SPKTAB3

The cursor is moved one position to the left. The acoustic alarm will be sounded if an attempt is made to go beyond the left hand field limit.

Insert - Position 3 in SPKTAB3

A space character is inserted into the string at the current cursor position, the characters to the right of the cursor being moved one place to the right. The character string will be truncated if it exceeds the field limit.

Delete - Position 4 in SPKTAB3

The character at the current cursor position is deleted, the characters to the right of the cursor will then be shifted left one place, the right most position being replaced by a space character.

Backspace - Position 1 in SPKTAB1 and SPKTAB2

When this key is pressed, the cursor is moved one place to the left and the corresponding position on the display is replaced by a period '.'. If backspace is performed at the first position of the current input field, then the previous contents of the data item, if any, will be displayed.

9.2.5.2 Clear functions

Clear 1 - Position 2 in SPKTAB1 and SPKTAB2, position 5 in SPKTAB3

The input field on the display unit and the corresponding data item will be cleared, the cursor is placed at the first position of the current input field. If this key is pressed when in EDIT mode the edit will be terminated and the next input field made current.

Clear 2 - Position 3 in SPKTAB1 and SPKTAB2, position 6 in SPKTAB3

The cursor is placed at the first position of the current input field, the previous contents of the data item, if any, will be displayed. If this key is pressed when in EDIT mode, the edit will be terminated and the next field made current.

Clear 3 - Position 7 in SPKTAB3

Characters from the current cursor position to the end of the field are deleted and the edit operation is terminated, the next field is made current.

End of item - Position 8 in SPKTAB3 and position 4 in SPKTAB1 and SPKTAB2

The current operation is completed, the next field is made current.

9.2.5.3 Cancel functions

Cancel 1 - Position 5 in SPKTAB1 and SPKTAB2, position 9 in SPKTAB3

A return is made to the application program, with the index item SPBINW2 containing the value one. Note no check is made on the contents of the field via SPAPPL etc..

Cancel 2 - Position 6 in SPKTAB1 and SPKTAB2, position 10 in SPKTAB3

This is the same as Cancel 1 but the index SPBINW2 will contain the value two.

9.2.5.4 Tabulation functions

Tab forwards - Position 7 in SPKTAB1 and SPKTAB2, position 11 in SPKTAB3

Make the next input item current. If there are no more input items on the screen then the cursor is re-positioned at the start of the current field. The next field can be on the same line as the current field, or it may be on a subsequent line. If there is more than one input field on a line, then the Tab forwards will make the left most field on the next line current.

Tab backwards - Position 8 in SPKTAB1 and SPKTAB2, position 12 in SPKTAB3

Make the preceeding input item current. If the current input field is the first on the screen then the cursor will be positioned at the start of the current field.

Tab home - Position 9 in SPKTAB1 and SPKTAB2, position 13 in SPKTAB3

Move the cursor to the first position of the first input field of the current screen format.

Tab left and down - Position 10 in SPKTAB1 and SPKTAB2, position 14 in SPKTAB3

Tabulate to the first input field on the next line. If no input field exists on the screen, below the position of the current screen then no action will be taken.

Tab left - position 11 in SPKTAB1 and SPKTAB2, position 15 in SPKTAB3

Place the cursor at the first character position of the left most field of the current line.

Tab right - Position 12 in SPKTAB1 and SPKTAB2, position 16 in SPKTAB3

Place the cursor at the first character position of the right most field of the current line.

Tab down - Position 13 in SPKTAB1 and SPKTAB2, position 17 in SPKTAB3

Move the cursor to the data item on the next line, which is in the position nearest the current input field. If the line has two fields equidistant from the current field then the left most of the two fields will be selected. If there are no fields below the current field on the screen then no action will be taken.

Tab upwards - Position 14 in SPKTAB1 and SPKTAB2, position 18 in SPKTAB3

Move the cursor to the data item on the preceding line, which is in the position nearest the current input field. If the preceding line has two fields equidistant from the current field then the left most of the two fields will be selected. If there are no fields above the current field on the display then no action will be taken.

9.2.5.5 Miscellaneous functions

Copy - Position 15 in SPKTAB1 and SPKTAB2, position 19 in SPKTAB3

A copy of the screen will be produced on the printer. This will have the effect of terminating input to the current field,

Duplicate - Position 16 in SPKTAB1 and SPKTAB2, position 20 in SPKTAB3

The contents of the field specified in the duplicate option will be moved to the current input field. If the option was not specified in the FKI instruction, error message number four in the format table SPFTBERR, will be displayed on the last line of the screen and the acoustic alarm sounded.

Edit - Position 17 in SPKTAB1 and SPKTAB2, position 21 in SPKTAB3

The keyboard will be set to edit mode. If the package is already in edit mode this instruction is ignored and the acoustic alarm sounded.

Enter - Position 18 in SPKTAB1 and SPKTAB2, position 22 in SPKTAB3

This causes a return to the application. If compulsory fields have not been filled, then an error message is produced, and the empty compulsory field nearest the top left corner is made current.

Application function keys - Positions 18 and above in SPKTAB1 and SPKTAB2, positions 22 and above in SPKTAB3

These are treated as an ENTER. On return to the application the index item SPBINW2 will hold the number of the application key in the following form:- if application key 1 was pressed then SPBINW2 will hold the value 4, if application key 2 was pressed then SPBINW2 will hold the value five, and so on.

9.2.6 Format table

The error messages produced by screen management are held as a series of one line format lists, the names of these format lists being given in the format table SPFTBERR. The first five positions of this table have a predefined meaning within screen management, subsequent positions being available for holding application related error messages. The meaning applied to the first five elements is given in the table below:-

Format table entries for error messages in screen management

Pos.	Use
1	Number of characters entered is less then that specified in MINL
2	Not used (available to appl)
3	I/O error (e.g. time out)
4	Illegal end of item key (dupl. key pressed but not specified in FKI)
5	Compulsory field still blank
6	And upwards available to appl.

When an error is detected either in the application, or screen management, the acoustic alarm will be sounded and the appropriate message displayed on the last line of the screen.

When an error message has been displayed on the screen - not via SPERR, a correct value may be entered in the errored field only after one of the following function keys has been pressed:- CLEAR 1, CLEAR 2, EDIT, CANCEL 1, CANCEL 2.

9.2.7 Tabulation validation routine

It is necessary to write a subroutine called SPTCHK. This routine is called if the field that has been tabulated to has the CTAB option set, it will be called before data can be entered to the field. Hence a checking program can be written to see if the field can be made current in the existing environment, providing a means of controling the cursor position. The result of this routine is held in SPBINW3; in the case of a correct tabulation this data item will contain zero, and if the tabulation is to continue it will contain a non zero value.

Tabulation check routine

Contents of SPBINW3	Action
Zero	Correct tabulation
Any other value	Tabulation will continue

9.2.8 Value check routines

Data items entered may be checked via a user written subroutine before the value is transferred from the DYKI input area (SPINPUT) to the data item identifier specified in the FCOPY or FMEL instruction. These subroutines are written by the user and are called:-

- SPAPPL
- SPCHK1
- SPCHK2
- SPCHK3
- SPCHK4
- SPCHK5
- SPCHK6
- SPCHK7

These routines must be present as ENTRY points in the application, as they are defined as EXTernal in the SCREEN module.

The SPAPPL subroutine will be called after the DYKI instruction, if the associated field descriptor (FKI) contained the APPL option. The APPL option is followed by a number between -32768 and 32767. When the routine is called, the number will be held in the data item SPBINW3, to be used in an indexed or ordinary branch.

The subroutines SPCHK1 through SPCHK7 will be called after the DYKI instruction but before the SPAPPL routine, if the associated field descriptor (FKI) contained the SCHK option. The SCHK option is followed by a number between 1 and 7. This number is used to call the appropriate routine.

It is possible to specify both APPL and SCHK routines in the same FKI, in which case the SPCHK routine will be called first, the SPAPPL routine will be called when a RETurn is encountered in the SPCHKx routine.

The data items passed from screen management to the application checking routine are:-

SPINPUT The string data item containing the data that has been input from the keyboard.

SPBINW1 A binary data item containing the number of characters transferred

SPBINW2 A binary data item containing the converted end of item key index in the key table

SPBINW3 A binary data item containing the value defined in the APPL option of the FKI command, hence a number between -32768 and 32767.

The output from the SPCHKx series of routines and SPAPPL is shown below. The data item SPINPUT may be altered by one of these routines, but if its length is changed, then SPBINW1 must also be altered.

The data items passed from the application checking routine to screen management are:-

SPINPUT The string data item containing the data that has been input from the the keyboard, and possibly changed by the routine.

SPBINW1 A binary data item containing the number of characters transferred to the screen management package in SPINPUT.

SPBINW2 A binary data item containing the converted end of item key index in the key table.

SPBINW3 A binary data item containing one of the values from the table shown below.

SPBINW4 A binary data item containing an index to the error message to be displayed. If no message is to be displayed, then this will contain the value zero.

Contents of SPBINW3	Action
Zero	The contents of the data item SPINPUT will moved to the data item of the current input field and displayed on the screen when the 'REWRT' option is specified in the FKI command. Screen management will continue according to the end of item key.
One	The contents of the data item will be displayed and moved to the data item of the current input field. Screen management will continue according to the end of item key held in SPBINW2.
Two	The data item is <u>not</u> moved to the current input field, the cursor is set to the begining of the current field, and input can be performed on this field.
Three	Error condition, the binary data item SPBINW4 will contain the index to the error message in SPFTBERR, when SPBINW4 contains zero no message will be displayed.

Example of the use of value check routines

```
FNL
FTEXT     'PLEASE ENTER YOUR USER CODE '
FKI       29,MAXL=4,SCHK=1
FCOPY     USER
FNL
FTEXT     'PLEASE ENTER THE CURRENT DATE '
FKI       33,MINL=6,SCHK=5,APPL=8
FCOPY     DATE
FMEND
```

When the user code is entered it is validated by the subroutine SPCHK1. The number of characters entered, excluding the EOI character, will be held in SPBINW1, the actual characters entered in SPINPUT. The transfer of characters from SPINPUT to USER takes place when the FCOPY instruction is encountered.

When the date is entered the subroutine SPCHK5 will be called first and then SPAPPL; when SPCHK5 is called, SPBINW4 will hold the value 5; when SPAPPL is called, SPBINW3 will hold the value 8. If an error was detected by SPCHK5 then the message would be displayed etc. without the SPAPPL routine having been called.

Note:

- If the number of characters in SPINPUT is changed, then SPBINW1 also must be changed to the new number.
- The end of item key index may be changed, for example the EOI code X'03' may be changed to the ENT code X'17'.

```
REL 4.1 790523 *  PROCEDURE DIVISION          * IDENT SCREEN  * DATE 791015 * PAGE 0002

LINE  LABEL     OPCODE   OPERANDS              COMMENT

0009            PDIV
0010  *
0011            ENTRY    SPCLRA                CLEAR ALL VARIABLE FIELDS
0012            ENTRY    SPCLRS                CLEAR SOME VARIABLE FIELDS
0013            ENTRY    SPCLRN                CLEAR NO VARIABLE FIELDS
0014            ENTRY    SPERR                 DISPLAY ERROR MESSAGE, UPDATE
0015                                           .. CURRENT FIELD & CONTINUE IN
0016                                           .. FORMAT.
0017            ENTRY    SPERR2                DISPLAY ERROR MESSAGE, UPDATE
0018                                           .. CURRENT FIELD & RETURN.
0019  *
0020            EXT      SPCHK1                STANDARD CHECK ROUTINE NO. 1
0021            EXT      SPCHK2                STANDARD CHECK ROUTINE NO. 2
0022            EXT      SPCHK3                STANDARD CHECK ROUTINE NO. 3
0023            EXT      SPCHK4                STANDARD CHECK ROUTINE NO. 4
0024            EXT      SPCHK5                STANDARD CHECK ROUTINE NO.5
0025            EXT      SPCHK6                STANDARD CHECK ROUTINE NO. 6
0026            EXT      SPCHK7                STANDARD CHECK ROUTINE NO. 7
0027            EXT      SPAPPL                USER ROUTINE TO HANDLE
0028                                           APPL VALUES
0029            EXT      SPTCHK                USER ROUTINE TO EVALUATE
0030                                           CONDITIONAL TABULATION
0031                                           APPL VALUES
0032                                           CONDITIONAL TABULATION
0033  *
0034            EXT      EMPTYT                ASSEMBLY SUBROUTINE EMPTYT -
0035                                           TEST IF DATA ITEM IS EMPTY
0036  *
```

```
LINE  LABEL    OPCODE  OPERANDS                    COMMENT                    C

0037           EJECT
0038           INCLUDE SPLITT,LIST
0000- *
0001- *◊S P L I T T
0002- *◊THIS MODULE IS USED BY SCREEN-PACKAGE.
0003- *◊IT CONTAINS KEYTABLES AND ERROR PRINTOUTS
0004- *◊USED IN CONNECTION WITH SCREEN PACKAGE
0005- *
0006- BSP      EQU     X'89'
0007- CLEAR    EQU     X'A3'
0008- CANC1    EQU     X'A2'
0009- CANC2    EQU     X'99'
0010- REWRT    EQU     X'95'
0011- EOI      EQU     X'90'
0012- FWD      EQU     X'86'
0013- BWD      EQU     X'85'
0014- HOME     EQU     X'88'
0015- LDOWN    EQU     X'8B'
0016- LEFT     EQU     X'81'
0017- RIGHT    EQU     X'8A'
0018- DOWN     EQU     X'87'
0019- UP       EQU     X'82'
0020- HCOPY    EQU     X'91'
0021- DUP      EQU     X'96'
0022- CFWD     EQU     X'A0'
0023- CBWD     EQU     X'A1'
0024- ENT      EQU     X'92'
0025- MINUS    EQU     X'97'
0026- NOKEY    EQU     X'FF'
0027- INS      EQU     X'84'
0028- DEL      EQU     X'83'
0029- CLR3     EQU     X'98'
```

```
LINE  LABEL    OPCODE  OPERANDS                COMMENT                    C

0030- TYPE     EQU     X'80'
0031- TEST     EQU     X'93'
```

```
LOC  OC OPERANDS            LINE  LABEL    OPCODE  OPERANDS                    COMMENT                C
                            0032-          EJECT
                            0033- *◊KEYTABLE USED WHEN CURSOR IS PLACED AT THE VERY
                            0034- *◊FIRST POSITION IN THE CURRENT FIELD
                            0035- *
                            0036- SPKTAB1  KTAB    BSP,CLEAR,REWRT,EOI,CANC1,CANC2,                C
                            0037-                  FWD,BWD,HOME,LDOWN,                             C
                            0038-                  LEFT,RIGHT,DOWN,UP,HCOPY,                       C
                            0039-                  DUP,CFWD,ENT,NOKEY,TYPE
                            0040- *
                            0041- *◊KEYTABLE USED WHEN CURSOR IS PLACED AT ANY
                            0042- *◊POSITION BUT THE VERY FIRST IN THE CURRENT FIELD
                            0043- *
                            0044- SPKTAB2  KTAB    BSP,CLEAR,REWRT,EOI,CANC1,CANC2,                C
                            0045-                  FWD,BWD,HOME,LDOWN,                             C
                            0046-                  LEFT,RIGHT,DOWN,UP,HCOPY,                       C
                            0047-                  NOKEY,CBWD,ENT,MINUS,TYPE,TEST
                            0048- *
                            0049- *◊KEYTABLE USED IN EDIT MODE
                            0050- *
                            0051- SPKTAB3  KTAB    CFWD,CBWD,INS,DEL,CLEAR,                        C
                            0052-                  REWRT,CLR3,EOI,CANC1,CANC2,                     C
                            0053-                  FWD,BWD,HOME,LDOWN,                             C
                            0054-                  LEFT,RIGHT,DOWN,UP,HCOPY,                       C
                            0055-                  NOKEY,NOKEY,ENT,MINUS,TYPE,TEST
                            0056- *
                            0057- *◊FORMAT TABLE CONTAINING ERROR PRINTOUTS
                            0058- *
                            0059- SPFTBERR FTABLE  ERFM01,ERFM02,ERFM03,ERFM04,ERFM05,ERFM06
                            0060- *
                            0061- ERFM01   FRMT
0000 E0                     0062-          FSL
0001 C3 18 ..               0063-          FTEXT   'TOO FEW INPUT CHARACTERS'
```

```
* CREDIT TRANSLATOR  REL 4.1 790523 *  PROCEDURE DIVISION        * IDENT SCREEN-  * DATE 791015

LOC  OC OPERANDS              LINE  LABEL    OPCODE  OPERANDS                COMMENT

                              0064-          FMEND
                              0065- *
                              0066- ERFM02   FRMT
0000 E0                       0067-          FSL
0001 C3 0F ..                 0068-          FTEXT   'UNDEFINED ERROR'
                              0069-          FMEND
                              0070- *
                              0071- ERFM03   FRMT
0000 E0                       0072-          FSL
0001 C3 09 ..                 0073-          FTEXT   'I/O-ERROR'
                              0074-          FMEND
                              0075- *
                              0076- ERFM04   FRMT
0000 E0                       0077-          FSL
0001 C3 17 ..                 0078-          FTEXT   'ILLEGAL END-OF-ITEM KEY'
                              0079-          FMEND
                              0080- *
                              0081- ERFM05   FRMT
0000 E0                       0082-          FSL
0001 C3 1B ..                 0083-          FTEXT   'COMPULSORY FIELD NOT FILLED'
                              0084-          FMEND
                              0085- ERFM06   FRMT
0000 E0                       0086-          FSL
0001 C3 0D ..                 0087-          FTEXT   'ILLEGAL VALUE'
                              0088-          FMEND
```

10. DATA COMMUNICATION

10.1 Introduction

The subject of data communication is a complex one, with different protocol systems for each kind of mainframe. For a programmer to arrange all the correct sequences of control instructions to go down the line at the right time time will obviously be a complex task. This is handled in CREDIT by having a number of differant drivers to handle the differing protocols.

There are two instructions used with data communication :-

READ	read data from the line
WRITE	send data down the line

The DC action is treated in exactly the same manner as any I/O operation. A dataset must be defined for the line and given a dataset name for use with the instructions. The action on the line is different between point-to-point and multipoint configurations, as shown below.

10.2 Time out

Because of the way in which DC applications work, it is necessary to incorporate a 'timeout' function. This means that a time limit is set for completion of the read or write request. If no message has been received on a read after the timeout expires, the request is completed, and an indication is given to the task of this occurrence.

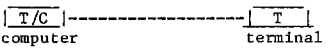
It is important that the terminal task and mainframe task should have different timeout values, to prevent a BID collision. This occurs when both the terminal and the mainframe are trying to use the line at the same time. If both requests time out and then try the request again, the same collision will occur. This can be avoided if the mainframe task has a shorter timeout value than the terminal task, thereby getting priority on the line.

To set the time out, which may be altered before each read or write, a data set control instruction is used:-

```
DSC1    DSDC,X'0B',TIME
```

10.3 Point-to-point

This is the simplest form of data communication with the computer connected to the terminal by means of a cable, see below:-



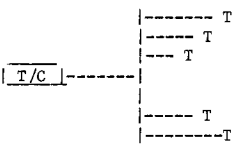
The two instructions described before are adequate for this situation. The terminal should be polled regularly in case it wants to send a message. This can be performed by issuing a read instruction at regular intervals. If an answer is required, the write instruction is performed to send data back to the terminal.

If the task is performing on a strict question and answer basis, it must be remembered that the task may still be switched between reading and writing, since this involves an LKM request.

If the terminal is connected to the computer via a telephone or telegraph line, then it may be necessary for an operator to dial the number and switch the modem or accoustic coupler on, when the carrier tone is heard.

10.4 Multipoint

This situation is more complex than point-to-point as a number of terminals could be connected to the line. this is shown in the diagram below:-



The line can be connected either permanently, or switched if all terminals are in the same area, so the connection is the same as described above. All messages must be addressed to the required terminal by transferring the terminal address to the DC driver. The same must be performed for a read instruction from a particular terminal, or by the terminal task when addressing a message to the computer.

10.5 DC task

With some applications it is possible for a terminal computer to receive data from the mainframe without it being requested. It is advisable in this case to write a task dedicated to the receiving of unsolicited messages. This task would have a READ outstanding on the line with no time out set. It would be entirely up to the application how the message was handed over to the processing task but this could be done using intertask communication.

11. PROGRAM DEVELOPMENT AND TESTING

11.1 Introduction

The source data for a CREDIT program may be entered to the system under DOS by one of three media:

- Cassette (standard assignment)
- Punched cards
- Console typewriter

The sequence of processes necessary to develop a program is shown on foil 42, and all these processes take place under DOS.

The testing of programs, however, takes place under control of the TOSS Monitor, as has been mentioned before.

The processors required for the development of a credit application are described in this section.

11.2 CREDIT Translator

The CREDIT Translator is called into execution by the TRA command, and performs the following actions:

- Each module is processed separately by the Translator.
- This produces an Intermediate Object Code module, which must be made permanent by the KPF command, unless it contains the DDIV for the entire application.
- The instructions in these modules use a byte-oriented addressing system, and this code is printed on the output listings at the left hand side.
- Each module may contain references to:
  - Labels in the same module
  - Literals in the same module
  - Labels in other CREDIT modules
  - Assembler application modules
  - Assembler system routines

The first type of reference is satisfied by the CREDIT Translator.

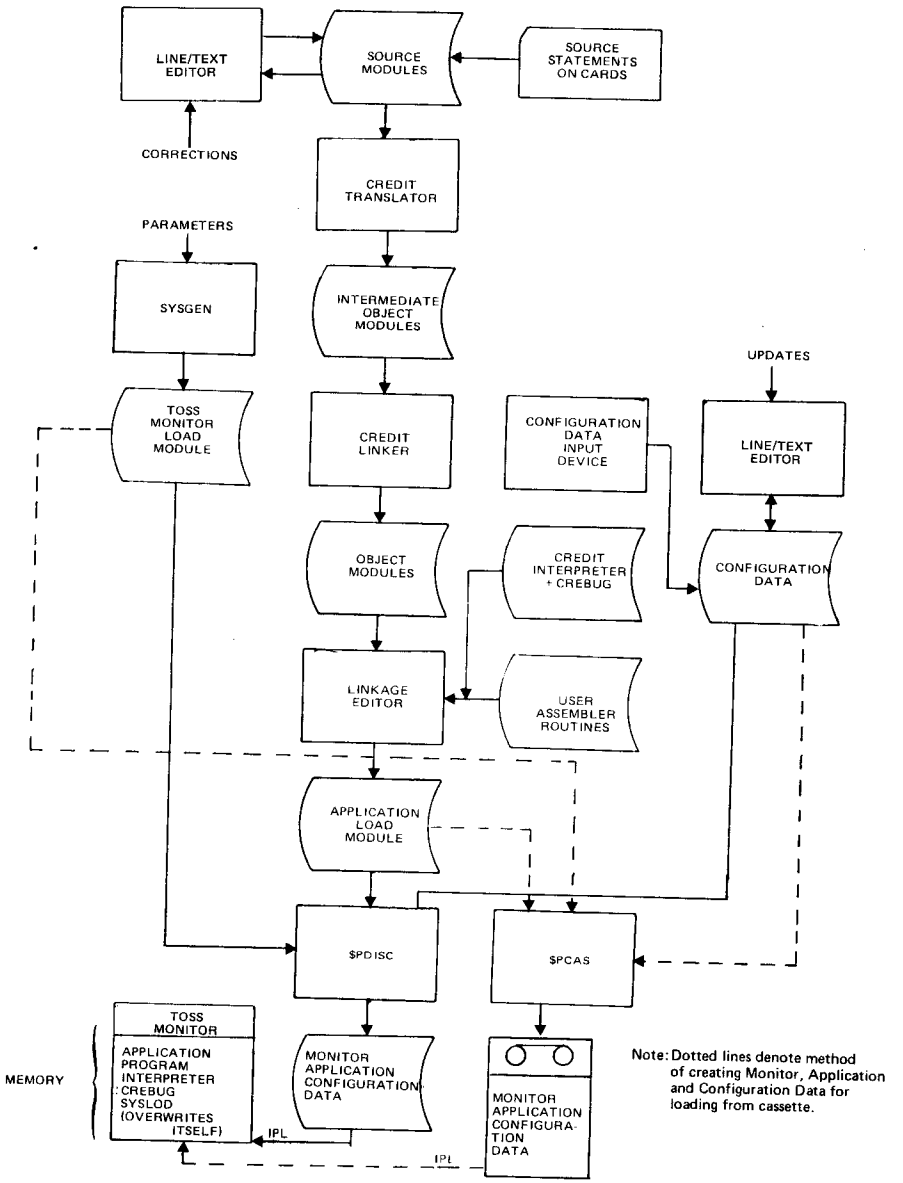
It is recommended that all temporary files be scratched before running the Translator.

11.3 CREDIT Linker

The resulting object modules are then processed by the CREDIT linker. This is called into execution by the TLK command, and performs the following functions:

- Solves references to the second two types of reference described above.
- Links together the object modules to form word-oriented object modules.

Keyword	Page in Manual
TLK	M11 6.12.43
TRA	M11 6.12.44
KPF	M11 6.12.12


LINE/TEXT EDITOR
SOURCE MODULES
SOURCE STATEMENTS ON CARDS
CORRECTIONS
CREDIT TRANSLATOR
PARAMETERS
SYSGEN
INTERMEDIATE OBJECT MODULES
UPDATES
TOSS MONITOR LOAD MODULE
CREDIT LINKER
CONFIGURATION DATA INPUT DEVICE
LINE/TEXT EDITOR
OBJECT MODULES
CREDIT INTERPRETER + CREBUG
CONFIGURATION DATA
LINKAGE EDITOR
USER ASSEMBLER ROUTINES
APPLICATION LOAD MODULE
$PDISC
$PCAS
TOSS MONITOR
APPLICATION PROGRAM INTERPRETER : CREBUG SYSLOD (OVERWRITES ITSELF)
MEMORY
MONITOR APPLICATION CONFIGURATION DATA
MONITOR APPLICATION CONFIGURATION DATA
IPL
IPL
Note: Dotted lines denote method of creating Monitor, Application and Configuration Data for loading from cassette.

11.3.1 Segmentation

It is possible for an application to be split up into a number of 'segments'. There is always one segment, number 00, which contains the DDIV, Interpreter, and Assembler subroutines. This segment is always resident in memory during application running. The remaining segments may be disk resident, or memory resident, depending on a) the wishes of the programmer, and b) the size of memory of the machine in use. If no action is taken by the programmer on segmentation, then the result of the TRA and TLK processing described above will be an unsegmented program, i.e. segment 00 will contain the entire application. If segmenting is required, the segments are built up by use of the INC and NOD commands.

11.4 Linkage Editor

The output modules from the CREDIT linker are processed by the Linkage Editor. This is called into execution by the LKE command, and perfoms the following functions:

- Links together all the modules from the Linker to form one application load module.
- Solves the remaining references between modules and system routines.
- Includes Assembler application routines if required.
- Includes the CREDIT Interpreter.
- Includes the CREDIT configuration program.
- Includes the CREDIT Debugger, unless explicitly excluded.

11.5 CREDIT Interpreter

The load module created by the above processors can not be executed directly by the machine, but must be in a format suitable for execution under the control of the TOSS Monitor. The way in which the moving from DOS to TOSS is explained below. Once the load module is in memory, it must be interpreted by the CREDIT Interpreter, that is the functions in the CREDIT intermediate object code are called into execution by the Interpreter by means of calls to Assembler system routines.

11.6 CREDIT Configurator

After system configuration, which is covered later, the CREDIT configurator takes control: this sets up all the required workblocks, stacks, data set buffers, and task control areas that are required by the tasks to be executed.

Following this, control is handed to the Interpreter, and the program commences execution.

Keyword	Page in manual
INC	M11 6.12.11
NOD	M11 6.12.20

11.7 CREDIT Debugger

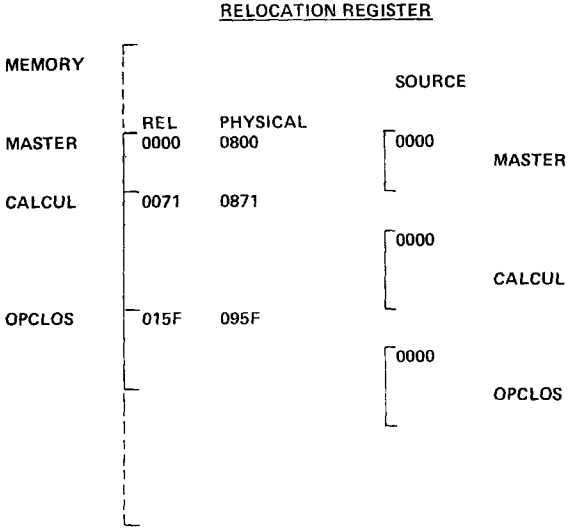
The CREDIT debugging program (CREBUG) is an interactive diagnostic task which runs under the control of the TOSS monitor. It is used to control execution of the application in the following way:-

- Traps may be set
- Variables may be examined and modified
- Trace may be turned off

etc.

CREBUG is specified as a special task at SYSGEN time; it runs at a priority leval higher than that of the application, to enable the application task to be interupted, it also has a special task identifier TB.
The programmer can use the Translator and linkage lists to set traps, verify the contents of data items change elements in the picture pool etc.

Keyword	Page in manual
Debugger	M04 4.1.1



REG.D	CONTAINS 0800
1Q REG.1	/0000 0071,D CONTAINS 0871
2Q REG.2	/0000 015F,D CONTAINS 095D

LOAD MAP

MODULE	LOC	ERROR
MASTER	0000	
CALCUL	0071	
OPCLOS	015F	
OPOPEN	0192	
READN	01E3	
SYCLOS	0254	
SYSOPN	0283	
BOOK	02E8	

MASTER	0000	APPLICATION PROGRAM
CALCUL	0071	
OPCLOS	015F	
OPOPEN	0192	
READN	01E3	
SYCLOS	0254	
SYSOPN	0283	
BOOK	02E8	

11.8 Line Editor and Text Editor

If errors do occur while testing, it is possible to correct some of them via the debugger. However, at some stage the source code has to be corrected or updated. This can be done easily by using one of the DOS processors, the Line Editor/Text Editor. By the use of various commands available, the source code can have lines amended, inserted and deleted.

Keyword	Page in manual
Line editor	M11 8.2.1
Text editor	M11 2.1.1

11.9 CREDIT Translator Listings

During the processing of the CREDIT Translator, a listing is produced (unless specifically suppressed), containing:

- CREDIT Source statements
- Intermediate Object code
- Error messages

The heading of the listing contains the Release Number of the Translator in use, and the date, the heading Data Division or Procedure Division, as appropriate, and the name of the module from the IDENT statement. Reading from left to right across the page, the following appear:

LOC = Location Counter

This is a four digit hexadecimal counter, which is increased by one every time a byte of intermediate object code is produced, for the Procedure Division only. This counter is used when using the CREDIT Debugger, to display and/or amend the contents of memory. For the data division, the counter is the index value of items within workblocks, where the first digit is the workblock number and the second the number of the item in the workblock. Thus 32 = Workblock 3, Item 2 (Workblocks start at 1, Items 0). Note that Boolean data items, for which one word is reserved in each block, the second number is the bit within the word at the start of the block. These numbers are also used for DSET statements, where they are again an index value of the DSET within the task.

OC OPERANDS

These are the Operation Code and the Operands generated from the CREDIT source, in Intermediate Object Code, they are printed at the left hand side of the translator listing.

Where an operand is shown as LL, this is a reference to the literal pool, which is filled in by the CREDIT Linker later. This code is present for Literals (=X'6142')
Format Lists (FRMT)
Keytables (KTAB)

Where an operand is shown as RR, this is a reference to a subroutine within the same module, which is also filled in by the Linker.

Where an operand is shown as XX, this is a reference to an external routine, also filled in by the Linker.

STMT = Statement

This is a four digit decimal line number, and is the line number used when editing with the Line Editor on the source module.

LABEL OPCOD OPERANDS COMMENT

These are self-explanatory.

C = Continuation

This shows that the statement is continued on a new line, as it was too long for one source line.

ERROR MESSAGES

If the Translator detects an error in the source code, it prints an explanatory message under the statement in error, together with an asterisk to indicate the part of the statement that is incorrect.

At the end of the listing, the messages PROGRAM LENGTH and ERROR are printed. The program length is the hexadecimal number of bytes contained in the module, and the error count is a decimal count of the number of errors detected in the module.

In addition, two tables are listed at the end:

- Data item name table, showing all the data item names used in the module, with a U printed by them if they are not referenced within this module.
- Procedure label table, showing the labels (names) of all the PROC statements referred to in the module.

```
• CREDIT TRANSLATOR  REL 4.1 790523 •  DATA DIVISION              • IDENT EX7    -· • DATE·790920 • PAGE 0001 •
```

```
      IX                LINE  LABEL    OPCODE  OPERANDS                  COMMENT
                        0000          IDENT   EX7
                        0001          DDIV
                        0002          TERM    A1
                        0003          CWB     CB1
      10                0004  DSKB    DSET    FC=20,DEV=KB
      11                0005  DSDY    DSET    FC=40,DEV=DY,BUFL=100
                        0006          START   GOA1
      1                 0007  CB1     BLK
      10                0008  INLEN   BIN
      11                0009  IX1     BIN
      12                0010  AMNT    BCD     11D
      13                0011  ACCNT   STRG    9
      14                0012  IBUF    STRG    11
```

```
• CREDIT TRANSLATOR  REL 4.1 790523 •  PROCEDURE DIVISION        • IDENT EX7   -- • DATE 790920 • PAGE 0002

LOC  OC OPERANDS              LINE  LABEL    OPCODE  OPERANDS                  COMMENT

                              0013          PDIV
                              0014          ENTRY   G0A1
                              0015  KTB1    KTAB    X'23'
0000 01 10 LL                 0016  G0A1    MOVE    INLEN.=W'10'
0003 30 XX D0 14 KK 10 11     0017          NKI     DSKB.IBUF.KTB1.INLEN.IX1
000A 5A 0C                    0018          BERR    G0A1
000C 36 11 01 02              0019          IB      IX1.NUMIN
0010 5F 12                    0020          B       G0A1
0012 19 10 LL 16              0021  NUMIN   CBNE    INLEN.=W'10'.G0A1
0016 00 13 14                 0022          MOVE    ACCNT.IBUF
0019 01 10 LL                 0023  RDAM    MOVE    INLEN.=W'11'
001C 30 XX D0 14 KK 10 11     0024          NKI     DSKB.IBUF.KTB1.INLEN.IX1
0023 5A 0C                    0025          BERR    RDAM
0025 36 11 01 02              0026          IB      IX1.AMTIN
0029 5F 12                    0027          B       RDAM
002B 19 10 LL 16              0028  AMTIN   CBNE    INLEN.=W'11'.RDAM
002F 00 12 14                 0029          MOVE    AMNT.IBUF
0032 31 XX 91 FF              0030          DSWRT   DSDY.OUTF
0036 5F 38                    0031          B       G0A1
                              0032  ******
                              0033  OUTF    FRMT
0000 C1 LL                    0034          FCOPY   =X'2031'
0002 C3 08 ..                 0035          FTEXT   'ACCOUNTNR. '
0005 C0 13                    0036          FCOPY   ACCNT
0011 E8                       0037          FEOR
0012 C1 LL                    0038          FCOPY   =X'2030'
0014 C3 08 ..                 0039          FTEXT   'AMOUNT     '
0021 PP 12                    0040          FMEL    '99999999V99',AMNT
                              0041          FMEND
                              0042          END
```

```
* CREDIT TRANSLATOR  REL 4.1 790523 *  PROCEDURE DIVISION          * IDENT EX7    -- * DATE 790920 * PAGE 0002 *

LOC  OC OPERANDS              LINE  LABEL    OPCODE  OPERANDS                  COMMENT                        r

PROGRAM LENGTH = 0038  ERROR = 0000
```

```
• CREDIT TRANSLATOR  REL 4.1 790523 •  DATA ITEM NAME TABLE        • IDENT EX7    • DATE 790920 • PAGE 0004 •
```

NAME	REF	TYPE	NAME	REF	TYPE	NAME	REF	TYPE	NAME	REF	TYPE
ACCNT	13	STR	AMNT	12	BCD	IBUF	14	STR	INLEN	10	BIN
IX1	11	BIN									

```
• CREDIT TRANSLATOR  REL 4.1 790523 •  PROCEDURE LABELS          • IDENT EX7  • DATE 790920 • PAGE 0005 •

NAME      REF   TYPE        NAME      REF   TYPE        NAME      REF   TYPE        NAME      REF   TYPE
AMTIN     0028  ADR         GOA1      0000  ADR         KTB1      0000  KEY         NUMIN     0012  ADR
OUTF      0001  FOR         RDAM      0019  ADR         T:EDWR    0002  EXT         T:NKI     0004  EXT
PROG ELAPSED TIME:  00H-00M-32S-760MS-
```

11.10 CREDIT Linker listings

The following listings are produced by the CREDIT Linker:

- Load map
- Long branch table
- Call table
- Perform table
- Literal pool
- Format pool
- Keytable pool
- Segment map
- Address cross reference list
- Literal cross reference list
- Picture/format cross reference list
- Linker statistics

11.10.1 Load map

This is used for setting the relocation registers when debugging programs.

LOC	MODULE	ERROR	COMMENT
0000	EX7		TRA 4.1 79-09-20 F1 01111

```
• CREDIT CODE LINKER REL 4.1 790523 • CALL TABLE  SEGMENT  00                - • DATE 790920 • PAGE    2 •
```

```
LOC   DATA      IX  SYMBOL  DEFINED

003A  ****      01  T:NKI
003C  ****      02  T:EOWR
```

11.10.2 Call table

This table contains all referances to external routines (CALL instructions) which were not satisfied by the TLK command. Each time a reference is encountered in the intermediate code, the linkage editor (LKE command), replaces it by an 'index value' which points to the called address in the call table. During execution of the application program, the interpreter refers to the call table for actual destination addresses.

- LOC is the displacement of each entry in the table within segment zero.
- DATA is the call Address relative to the start of segment zero.
- IX is the index value (01 - FF).
- SYMBOL is the name of the external routine.
- DEFINED is not used in this table.11.10.3 Long branch table

• CREDIT CODE LINKER REL 4.1 790523 • LE TABLE SEGMENT 00 • DATE 790702 •

LOC	DATA	IX	SYMBOL	DEFINED
0DA4	00 0037	01	GOMED2	MED237
0DA8	00 004E	02		MED237
0DAC	00 0068	03		MED237
0DB0	00 0051	04		MED237
0DB4	00 0AB5	05		MED237
0DB8	00 0CCC	06		MED237
0DBC	00 02DA	07		MED237
0DC0	00 00F4	08		MED237
0DC4	00 000F	09		MED237
0DC8	00 06B0	0A		MED237
0DCC	00 0152	0B		MED237
0DD0	00 0106	0C		MED237
0DD4	00 0008	0D		MED237
0DD8	00 06F1	0E		MED237
0DDC	00 0195	0F		MED237
0DE0	00 016F	10		MED237
0DE4	00 0846	11		MED237
0DE8	00 01D2	12		MED237
0DEC	00 0083	13		MED237
0DF0	00 0266	14		MED237
0DF4	00 020D	15		MED237
0DF8	00 02F8	16		MED237
0DFC	00 03E5	17		MED237
0E00	00 03E9	18		MED237
0E04	00 0316	19		MED237

11.10.3 Long branch table

In order to reduce the ammount of memory required for a long branch instruction, the linker (TLK) generates a table of destination addresses. Each time a long branch instruction is encountered in the intermediate code, the linker places the destination address (i.e. segment nimber and the address to be branched to) in the long branch table.

The three byte destination address in the long branch instruction is replaced by a one byte 'index value' which points to the destination address in the long branch table. During execution of the application program the interpreter refers to the long branch table for actual destination addresses.

- LOC is the displacement of each entry in the segment.
- DATA is the destination address and segment number.
- IX is the index of the entry in the table, and starts at the first number after the last number for the same type of table in segment zero; this applies to all tables.
- SYMBOL is the first instruction in the module containing the destination.
- DEFINED is the module containing the destination.

```
--------------------------------------------------------------------
* CREDIT CODE LINKER REL 4.1 790523 * PERFORM TABLE    STATEMENT  00
--------------------------------------------------------------------

LOC    DATA       IX  SYMBOL  DEFINED

0E8A   00 0284    01          MEDIS*
0E8E   00 07E4    02          MEDIS*
```

11.10.4 Perform table

This table contains the address of each CREDIT subroutine which is called (PERF or PERFI instructions) within this segment. It has the same layout as the long branch table. Each time a perform to a CREDIT subroutine is encountered, in the intermediate object code the subroutine name is replaced by an 'index value' which points to the subroutine address in the perform table.

- LOC is the displacement of each entry in the segment.
- DATA is the destination address.
- IX is the index of the entry in the table.
- SYMBOL is the name of the subroutine.
- DEFINED is the name of the module containing the subroutine.

```
---------------------------------------------------------------------------
• CREDIT CODE LINKER REL 4.1 790823 • LITERAL POOL  SEGMENT  00
---------------------------------------------------------------------------

IX    TYPE    LOC   DATA

10    BIN     003E  000A
11    BIN     0040  0008
12    STR     0042  2030
13    STR     0044  2031
```

11.10.5 Literal pool

The literal pool contains all the literals used in this segment. Each time a literal is encountered in the intermediate code is replaced by an 'index value' which points to the literal in the literal pool.

- IX is the index value of the entry (01-FF or 4100-41FF).
- TYPE is BIN, BCD or STR.
- LOC is the displacement of the literal within the segment.
- DATA is the hexadecimal representation of the literal.

```
* CREDIT CODE LINKER REL 4.1 790523 * PICTURE POOL  SEGMENT  00

IX    TYPE    LOC   DATA

10    PIC     0046  39393939393939393939
```

11.10.6 Picture pool

The picture pool contains all picture strings used in this segment. Each time a referance to a picture string is encountered in the intermediate code, it is replaced by an 'index value' which points to the picture string in the pool.

- IX is the index value of the entry (01-FF or 5100-51FF).
- TYPE indicates that the entry is a picture string (PIC).
- LOC is the displacement within the segment.
- DATA is the hexadecimal representation of the picture string.

```
* CREDIT CODE LINKER REL 4.1 790523 * KEYTABLE POOL   SEGMENT  00

 Y    TYPE    LOC   DATA

10    KEY     0051  0123
```

11.10.7 Keytable pool

The keytable pool contains all keytables used in the [illegible] in segment zero. Each time a referance to a keytable is [illegible] intermediate code, it is replaced by an 'index value' [illegible] keytable in the pool.

- IX is the index value of the entry [illegible]
- TYPE indicates the entry is a keytable [illegible]
- LOC is the displacement of the [illegible]
- DATA is the hexadecimal represent[illegible]

```
* CREDIT CODE LINKER REL 4.1 790523 * FORMAT POOL   SEGMENT  00                 * DATE 790920 * PAGE     5 *
```

```
IX    TYPE    LOC   DATA
10    FMT     0053  C113C30B4143434F554E544E522E20CD13E8C112C30B414D4F554E5420202020201012
```

11.10.8 Format pool

The format pool contains all format lists used in the segment. Each time a referance to a format list is encountered in the intermediate code, it is replaced by an 'index value' which points to the format list in the pool.

- IX is the index value of the entry (00-FF or 7100-71FF).
- TYPE is FMT for a Format list or FTB for a Format table.
- LOC is the displacement withtin the segment.
- DATA is the hexadecimal representation of the list or table.

```
* CREDIT CODE LINKER REL 4.1 790523 * SEGMENT MAP

   S E G M E N T                    N U M B E R   O F
NUMBER   TYPE   LENGTH   USAGE     MODULES   ERRORS

00         C       162                 1        0
```

11.10.9 Segment map

This map gives a listing of the number of segments, the number of modules in each segment, and the number of bytes per segment.

```
---------------------------------------------------------------------
* CREDIT CODE LINKER REL 4.1 790523 * CROSS REFERENCE LISTING
---------------------------------------------------------------------

SYMBOL   TYPE  VALUE     SEG-DEFINED.   REFERENCES

GOA1       S   00 0000   00-EX7
T:EDWR     C                            00-EX7     (1)
T:NKI      C                            00-EX7     (2)
```

11.10.10 Linker statistics per segment

The format of the linker statistics listing per segment, the contents of the listing are self-explanatory.

11.10.11 Address cross reference listing

This listing provides cross reference between statement/subroutine identifiers in the PDIV, and the modules/segments in which they are referenced.

11.11 Linkage Editor listings

The first page of this listing shows all the tables and routines used by the interpreter that are linked into the application, and their start addresses.

The second page of this listing shows all the symbols (in this case start points) used within the routines listed on Page 1.

Example: T:NKI has an address within the range of T:IO, and is a start point within the Input/Output driver module.

```
0008  S:GTAB
0014  T:AA1D    TRA 4,2
0052  T:DA1D    TRA 4,1
006E  T:ATAB    TRA 4,4
0072  U:BTAB    TRA 4,3
00C4  S:BTAB    TRA 4,1
00E6  C:CB1D    TRA 4,4
0098  D:CB1D    TRA 4,3
00AE  P:MTAB
00E2  P:PIL
0184  I:NTP     T,REL=4,3
042A  I:EVR     T,REL=4,3
0730  I:ADS     T,REL=4,1
0896  I:CMP     T,REL=4,1
0918  I:CFA     T,REL=4,
0994  I:MOV     T,REL=4,1
0B9A  I:MUL     T,REL=4,1
0D46  I:DIV     T,REL=4,1
0F0E  I:NIF     T,REL=4,
0FA6  I:EDT     T,REL=4,1
138E  I:EDTE    T,REL=4,
145C  I:EDS     T,REL=4,
14EC  I:CTR     T,REL=4,1
1580  I:EVS     T,REL=4,1
15C2  I:TRM     T,REL=4,1
1648  I:EAR     T,REL=4,1
16BE  I:MEA     T,REL=4,1
1750  I:CHK     T,REL=4,
17E0  TRMCHK    T,REL=4,
1858  TRMTAB    T,REL=4,1
18F4  TRNBUG    T,REL=4,
3DF6  T:IO      T,REL=4,1
3670  T:LSEG    T,REL=4,1
```

••• SYMBOL TABLE •••

C:CB10	007A	R	D:CB10	009A	R	G0A1	00E2	R	I:ADD	0730	R	I:BUF	1684	R
I:CHK	1880	R	I:CMP	0896	R	I:CPA	0918	R	I:CPY	155E	R	I:DIV	0046	R
I:DLT	1482	R	I:EBR	1430	R	I:EBRB	1434	R	I:EBRI	1404	R	I:ECB	1678	R
I:ECPY	1180	R	I:ECTR	1196	R	I:ECW	1102	R	I:ECWC	110C	R	I:ED1	1054	R
I:ED5	145C	R	I:EDSK	101E	R	I:EDT	1050	R	I:EDW	1056	R	I:EFIL	11E6	R
I:EFLA	10D2	R	I:EGTB	1450	R	I:EGTD	144E	R	I:EINH	138E	R	I:EKIH	138E	R
ENL	1182	R	I:EOR	11C8	R	I:EPIC	1202	R	I:ER10	1688	R	I:ER11	168C	R
I:ER12	1690	R	I:ER13	1694	R	I:ER14	1698	R	I:ER15	169C	R	I:ER16	16A0	R
I:ER17	16A4	R	I:ER18	16A8	R	I:ER19	16AC	R	I:ER1A	1680	R	I:ER1B	16B4	R
I:ER1C	16B8	R	I:ERR	168C	R	I:ERR0	1648	R	I:ERR1	164C	R	I:ERR2	1650	R
I:ERR3	1654	R	I:ERR4	1658	R	I:ERR5	165C	R	I:ERR6	1660	R	I:ERR7	1664	R
I:ERR8	1668	R	I:ERR9	166C	R	I:ERRA	1670	R	I:ERRB	1674	R	I:ERRC	1678	R
I:ERRD	167C	R	I:ERRE	1680	R	I:ERRF	1684	R	I:ESKI	1446	R	I:ESL	11A0	R
I:ETAB	1168	R	I:ETXT	1180	R	I:EVA0	04EC	R	I:EVA1	044A	R	I:EVA2	0442	R
I:EVA3	044E	R	I:EVA5	0458	R	I:EVA6	0452	R	I:EVA7	045C	R	I:EVB	06F6	R
I:EVI	06E4	R	I:EVIN	13C8	R	I:EVNO	1448	R	I:EVS1	1650	R	I:EVS2	1668	R
I:EVT	0FE8	R	I:EVTO	06A2	R	I:EXIT	1136	R	I:FFSN	1110	R	I:FML	068C	R
I:HEX	1894	R	I:HEXB	188E	R	I:HEXW	1894	R	I:INS	1500	R	I:MCH	14A4	R
I:MOV	0994	R	I:MUL	0B9A	R	I:MVC	0982	R	I:NTFA	0F6E	R	I:NTP	01F4	R
I:NTPA	01D4	R	I:NTPR	01FE	R	I:NTR	16F6	R	I:PRT	181C	R	I:RTO	0260	R
I:RTA	025e	R	I:RT2	0248	R	I:SHIL	0D14	R	I:SH1R	0F50	R	I:SUB	0736	R
I:TRA	17A4	R	I:TRAO	179E	R	I:TR6	17EC	R	I:TRC	0102	R	I:XCP	155E	R
I:BM5	00E2	R	P:END	0184	R	P:MTAB	00AE	R	S:B1AB	0006	R	S:G1AB	0008	R
I:AA10	004A	R	T:A1AB	0070	R	T:BAI	0120	R	T:CAT	011A	R	T:CSE6	3674	R
I:DA10	0052	R	T:DSCO	3144	R	T:DSC1	3156	R	T:DSC2	3166	R	T:LD10	32FA	R
LDWE	32BA	R	T:EDWF	32BE	R	T:EDWI	3206	R	T:EDWR	3240	R	T:FDSO	3392	R
I:FDSP	3396	R	T:FMI	01AU	R	T:GUSP	3396	R	T:GTCW	3364	R	T:I01	3214	R
T:I02	3218	R	T:I03	321C	R	T:I04	3220	R	T:IORE	323A	R	T:KEY	017C	R
I:KI	30F6	R	T:KIAA	366C	R	T:KIAC	366E	R	T:KIPR	3316	R	T:LIT	0174	R
T:LOFS	36B4	R	T:LOPS	36BE	R	T:LSEG	3670	R	T:MWAI	31CA	R	T:NKI	30FA	R
T:PAT	0120	R	T:PIC	0178	R	T:READ	3128	R	T:STCW	3468	R	T:WAIT	31B4	R
T:WRIT	3104	R	T:XSTA	31A4	R	TB:ABT	1FBE	R	TB:ENT	1FCA	R	TB:RDC	2082	R
TC:ABT	1CEA	R	TC:CHK	1BEC	R	TC:ERR	1D02	R	TC:RDC	1CF6	R	TT:HLT	1E66	R
TT:LOP	0040	A	TT:MSG	1E70	R	TT:PRC	1E6A	R	TT:SNO	1C6E	R	TT:TID	1E6C	R
TT:TON	1D5A	R	TT:TRP	1D5C	R	TT:TRS	0042	A	TT:VER	1DE2	R	TT:VMM	0040	A
TT:VON	1DE0	R	U:BTAB	0074	R									

11.12 SYSLOD (Configuration data)

Two items are required in addition to the application load module for the execution of the program, these are:-

- The TOSS Monitor.
- Configuration data.

At system start, the Monitor is read into memory, followed by the application, and then, before the application is started, the system configuration program SYSLOD is executed. This performs the configuration of the system for the specific environment in which the application is to run.

Keyword	Page in manual
SYSLOD	MO4 3.4.1