PHILIPS

PHILIPS TERMINAL SYSTEMS

User Library

CREDIT-PROGRAMMER'S REFERENCE MANUAL

Module M04

Data Systems

PREFACE

Part 1 of this Manual describes the use of the CREDIT programming language in the Philips Terminal System. Part 2 of the Manual describes the processing and testing of CREDIT programs under the DOS 6800 System Software and TOSS System Software.
This manual is based upon release 4.1 of the CREDIT language.
The use of those parts of DOS 6800 System Software designed specifically for CREDIT programs is described in this Manual. Information concerning the use of the general purpose components of DOS 6800 System Software is contained in the DOS 6800 System Software PRM (M11). Readers of the present Manual are expected to be familiar with the contents of the DOS 6800 System Software PRM. The testing and production running of CREDIT programs is done under TOSS System Software. Information concerning TOSS System Software which is relevant to the writing, testing and running of CREDIT programs is included in the present Manual. Details about Data Management and TOSS utilities are described in the manuals M07 and M08 respectively.

CONTENTS

1 THE CREDIT LANGUAGE

1.1 Introduction

1.1.1 *General*

The CREDIT programming language has been developed specifically for the Philips Terminal System. It is an interpretive language.
The object code generated from CREDIT is executed via an Interpreter. A CREDIT application program is normally subdivided into a number of modules, each module containing the statements necessary to perform a logically discrete processing step.

Modules are written and translated separately. Translation is the process of converting CREDIT source statements into intermediate object code.
A CREDIT module is composed of three types of statement:

Directives
Declarations
Instructions

Directives direct the CREDIT Translator during the production of intermediate object code. They are not translated into object code but provide a framework within which the programmer codes his program module.
Declarations are used to specify the type, length and value of data items used as operands in the module. They are also used to define the interface between the application program and the rest of the PTS System.
Instructions direct the input, processing and output of data. That is, they specify the functions to be carried out by the computer and direct the sequence of events.
Declarations and instructions are translated into the data and instructions which comprise the object program.

1.1.2 *Terminal/Application Program Interface*

1.1.2.1 *Programs*

CREDIT applications programs are developed under DOS 6800 System Software. However, they can be run only under TOSS System Software.
Under TOSS System Software only one application program can be held in memory. Hence, all the application processing for a PTS System is normally incorporated into one application program (which may, of course, be subdivided into modules).
When the total size of the TOSS-monitor plus the CREDIT application exceeds 64K bytes, different possibilities exist to run such applications on the PTS range of computers.

a) For systems having a maximum main memory capacity of 64K bytes, the only possibility is to use secondary memory (disk, flexible disk). From this secondary memory segments of the application are loaded into main memory at runtime, when necessary. This is under control of the memory management software.
b) For systems having an extended main memory, memory addressing upto 256K bytes, the whole application can be placed in main memory. Extended main memory may also be combined with use of secondary memory.
A hardware feature, the memory management unit (MMU), enables memory addressing up to 256K bytes. This virtual storage technique is implemented, with using CREDIT memory management software

1.1.2.2 *Data Sets*

A data set is a reference to an input/output device or diskfile on which an application program may perform input/output operations. More than one data set may be configured in a single device. For example, a journal printer, tally roll printer and front feed printer are combined in the PTS 6221 Teller Terminal Printer. However, separate input/output operations can be performed on each of the three data sets.

1.1.2.3 *Terminal Classes*

In a PTS System there is normally a device configuration at each of several work positions. Each device configuration comprises one or more devices. Some work positions may have the same type of device configuration; e.g. bank tellers would normally all use the same type of configuration. There are normally other work positions with different configurations. 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 types of transaction, identical program code is used to service each of these work positions.

1.1.2.4 *Tasks*

The CREDIT language enables the programmer to utilize the same set of CREDIT statements for each work position in a terminal class.
This is achieved in the following manner.
The interpretive object code generated from CREDIT programs is re-entrant. This means that a number of independent tasks can be achieved, all executing a single copy of the application program. Each time data is sent from a work position a task is activated by the TOSS Monitor. Thus, several tasks can be active at the same time for a number of terminal classes.
The TOSS Monitor schedules the various tasks so that, at any time, several tasks may be waiting for input/output to be completed, whilst other tasks are queued waiting for execution. Though only one task may be executed at a given instant, the overall impression is that all work positions are being serviced simultaneously.
Each task is assigned a unique task identifier by the system. This identifier is derived from the task identifier assigned to each terminal class by the programmer. With extended main memory, the TOSS-monitor always resides in the first 64Kbytes of main memory.

1.1.2.5 *Work Blocks*

One or more work blocks must be assigned by the programmer to each terminal class. These work blocks define areas of memory which may be used as working storage for e.g. input/output buffers. Dummy work blocks redefine these areas of memory. Swappable workblocks are stored on disk and will only on request be loaded into main memory.

1.1.3 *Program Design*

1.1.3.1 *General*

It is recommended that CREDIT programs be subdivided into modules. Each module should contain the statements necessary to perform a logically discrete processing step. There must be one main module in each program. This module will contain a complete data division headed by the DDIV directive. The remaining modules must not define a data division.

They should contain, instead, a DDUM directive followed immediately by the procedure division directive PDIV.
The result of this is that a single date division will be used by all modules in the program.
At least one terminal class should contain a program start point definition.
The remaining modules of the program may contain the statements required for the various types of transaction which the program is designed to process.
It is recommended that each module be devoted to the processing of a single transaction type.
It is the responsibility of the programmer to identify individual transaction types within a terminal class. This can be accomplished, for example, by testing a transaction code keyed-in at the work position by the user.
CREDIT programs may call subroutines written in PTS Assembler.
Certain system functions can be utilized only via Assembler programs. So it may be necessary to write a mixed CREDIT/Assembler program. However, the main module must always be written in CREDIT.

1.1.3.2 *Disk Resident Programs*

This way of extending the memory will lead to a decreasing of the performance, compared to memory extension with the memory management unit.

The code part consist of program segments just as for extended main memory. However, here the number of memory pages are not sufficient to permit all segments to be loaded in main memory together. The tasks have, as for other type of system resources to compete for main memory. The memory page replacing technique used is the least recently used method. This indicates that when the load in the computer goes down e.g. only a few tasks are running, these tasks will get a relatively large amount of main memory each. In situations of heavy computer load the tasks will get only the amount of main memory that is absolutely necessary. The dynamic allocation of main memory, when the system condition change, is controlled and supervised by the operating system itself. When looking at the code part it is important to consider the fact that the different tasks are using the same code to a great extent. In almost every application some or a lot of tasks are doing the same work on different physical work stations. These tasks are running the same instruction sequences but they are working on different and partly unique data areas. The situation above is valid for terminal systems in general. However the memory management technique is designed to handle also systems where the work within the system is delegated to a number of unique "specialist" tasks, each of them running its own program sequence. The difference will be that the competition for main memory will be harder in last-mentioned cases.

If no MMU is present the page size can be chosen to every value between sector size, 400 bytes, to 64K bytes, (also for flexible disk). The segmentation of the code part is made at linking time and the segments consists of; interpretable code, literal pool and address tables of the segment. Branches and subroutine calls will be solved automatically invisible to the user. Every segment can be loaded anywhere in main memory (in a page) and this decision is made by the system exclusively. Actually the only thing the user has to do is to define the program segment size.
The fetch policy used is, to load the segment at the point of time when it is needed, since it is very difficult to predict what segments will have to be loaded in a near future.

The replacement policy used e.g. the decision of which segment to overload when a new segment has to be loaded, is the Least Recently Used Method (LRU). A queue is built up telling which page in main memory to be replaced next time. This queue will be dynamically updated by the system each time a task is reactivated.

Note that the method described above implies that no dead-locks can appear, since there is always place for a new segment in main memory.

To take care of error situations (disk not operable, segment impossible to read) a special entry is defined in the resident part of the application program. (REENTER).

This virtual memory solution gives enough flexibility to the programmer to optimize the program execution. The most important thing is the concept of locality. When writing an application for a virtual memory system, the programmer should try to pack the frequently used modules to as small number of resulting segments as possible. In practice the following things can for example be considered when writing an application:
- remove exception and error-handling routines from the main path of the program.
- put all low Use routines in segments on their own.
- routines should be placed close to the routines they call or are called by.

Following rules should be noted, improving the locality of the program:
- there will be empty areas in the end of the pages, due to impossibility to make all the segments to the same size. The programmer, however, has the possibility to keep these empty areas at a low level.
- to have the possibility to build up and restructure the program segments, the programming technique to be used should be strongly modular.
- literals are placed in the segments where they are used.

1.1.3.3 *Extended Main Memory (up to 256K bytes)*

When using extended main memory, a special hardware feature the memory management unit (MMU) must be present.
The page size in systems with memory management unit (MMU) may be chosen by the user and should be a multiple of 1K bytes. This hardware feature allows a very fast paging system compared to disk as paging device. The page size is selected during linking. (TLK command, see chapter 2.3). The same rules are valid as mentioned for disc resident applications.

1.1.3.4 *Extended Main Memory and Disk Resident Programs*

The same rules are valid as mentioned for disk resident applications. The memory page replacement technique used, is the Least Recently Used Method, which will guarantee that the memory pages most frequently used will most of the time be situated in main memory.

1.1.4 *Source Input Format*

A CREDIT source program can be read into the PTS 6800 System using one of a variety of source input devices. Regardless of the input device used, the source data must have the following form.
A source line is an 80-character card image. If the input device allows records of variable length (console typewriter) each record must contain no more than one source statement. Input records longer than 80 characters are truncated, whereas shorter records are augmented by spaces up to column 80.

The source line is subdivided into four fields: label field, operation field, operand field and comments field. The label field begins in column 1. The label, operation and operand fields are each terminated by a tabulation character (\) or at least one space each. The operand field extends at maximum to column 71. If there are no non-space characters following the label (if any) before column 30, the rest of the statement is interpreted as a comment. Columns 73–80 are ignored in the translation process.
An asterisk in column 1 indicates that the source line is a comment. A source line containing spaces in columns 1–71 is ignored.
If column 72 contains a "C", the next line is interpreted as a continuation. For fixed length input records, the operand field may be terminated by a comma (leaving spaces up to column 72), the next operand starting on the continuation line. If a value inside quotes is split between two lines, all columns up to 72 are significant. For variable length records the operand field is terminated by two tabulation characters followed by a "C" for continuation. In this case, the character positions from the first tabulation character up to column 71 are not significant, and the operand field is immediately continued on the next line.
In continuation lines, the label and operation fields should be empty.

1.1.5 *CREDIT Syntax Definition*

The following symbols (Backus/Naur-Form) are used to define the syntax of CREDIT statements:

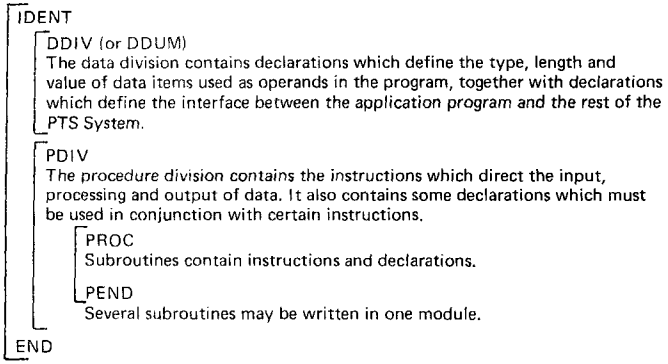
:: =	is defined as
␣	space
[ ]	the syntatic items between these square brackets may be omitted
{ }	select one of the items between the braces
a|b	select either a or b. This has the same meaning as braces. It is used with long strings.
...	ellipsis indicates that the last syntatic item may be repeated.

These symbols are used throughout the manual to define the syntax of CREDIT statements. They are also used in the parameter syntax definition in appendix 1.

1.2 Directives

1.2.1 *Structure Directives*

The framework of a CREDIT module is constructed from the directives IDENT, DDIV, DDUM, PDIV, PROC, PEND and END. The use of these directives is illustrated below:



The IDENT and END directives define the start and end of a module. They must be the first and last statements, respectively, of the module.
The DDIV or DDUM directive defines the start of the data division. It must be the second statement in the module. DDIV is used in the main module of a program. DDUM is used in all other modules.
The PDIV directive defines the start of the procedure division. The PROC and PEND directives define the start and end of a subroutine.
The IDENT, DDIV (or DDUM), PDIV and END directives must appear once only in each module. The PROC and PEND directives may be repeated. However, subroutines may not be nested. That is, two or more PROC directives cannot be written without an intervening PEND directive. Subroutines may, though, perform other subroutines.

1.2.2 *Linkage Directives*

CREDIT modules which have to be linked into an application program contain references to statements or subroutines in other modules. In order to achieve the correct inter-module linkages, entry points and external references must be specified. The ENTRY and EXT directives are used for this purpose. They must be written in the procedure division.
There must be at least one START directive for each program. When the TOSS System is started (i.e. the TOSS Monitor is loaded and the application program begins execution) a task is activated for each work position in the System. The tasks are activated at the start points specified in the START directives of the relevant terminal classes. The START directive(s) must be written in the data division and must be specified as entry points (ENTRY).

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 start points become pending. They will be activated after the first task has executed an EXIT instruction. When using memory management, the REENTER directive refers to a closing routine in case of a page fault or read error on disc. The statement identifier in this directive must be declared as ENTRY in the module it is defined.

1.2.3 *Listing Directives*

The directives LIST, NLIST and EJECT are used to control the CREDIT listing during the translation process. They may be written in any part of the module. The OPTNS directive must follow after the DDUM or DDIV directive.

1.2.4 *Equate directive*

This directive is used to equate an identifier with a value. When this identifier is used in an instruction the Translator automatically replaces it with the specified value, i.e. the instruction is translated as if the programmer had actually written the value in the instruction.
The directive may be used free in the procedure division (PDIV), but it should follow the ENTRY and EXT directives.

1.2.5 *Parameter directive*

The directives PFRMT, PKTAB and PLIT define the type of formal parameter, declared in the heading of the subroutine and must be used in two byte addressing mode or when a format table is used as formal parameter, or when in one byte addressing mode the literal constant name, keytable name or formatlist name, does not begin with a $ sign. These directives follow the PROC directive immediately.

1.2.6 *Directive reference*

This section describes the syntax and use of each directive. The possible values of the variables in the directives is given in appendix 1. The notation conventions are described in section 1.1.5.

DDIV *Data Division* DDIV

Syntax: ␣ DDIV ␣ [module-name]

Type: Structure directive

Description: Indicates the beginning of the data division of a module and causes a page feed in the listing during translation.

If module-name is specified, the data division statements will be fetched from the module indicated by module-name.

However, these data divisions will be entirely separate at execution time. The use of *common* data divisions is achieved through the DDUM directive.

No more than one module in a CREDIT program may contain a DDIV directive. The remaining modules must use the DDUM directive.

DDUM *Data division dummy* **DDUM**

Syntax: ␣ DDUM ␣[module-name]

Type: Structure directive.

Description: Beginning of a data division is indicated and the data division statements will be fetched from the module indicated by module-name.

No object code is output during processing of the data division.

Note: A DDUM-module may contain declaration of data items. The only difference between DDUM and DDIV directives is, when DDIV is declared the object modules of the data division are output on a /0 type file.

1 THE CREDIT LANGUAGE

1.1 Introduction

1.1.1 *General*

The CREDIT programming language has been developed specifically for the Philips Terminal System. It is an interpretive language.
The object code generated from CREDIT is executed via an interpreter. A CREDIT application program is normally subdivided into a number of modules, each module containing the statements necessary to perform a logically discrete processing step.

Modules are written and translated separately. Translation is the process of converting CREDIT source statements into intermediate object code.
A CREDIT module is composed of three types of statement:

Directives
Declarations
Instructions

Directives direct the CREDIT Translator during the production of intermediate object code. They are not translated into object code but provide a framework within which the programmer codes his program module.
Declarations are used to specify the type, length and value of data items used as operands in the module. They are also used to define the interface between the application program and the rest of the PTS System.
Instructions direct the input, processing and output of data. That is, they specify the functions to be carried out by the computer and direct the sequence of events.
Declarations and instructions are translated into the data and instructions which comprise the object program.

1.1.2 *Terminal/Application Program Interface*

1.1.2.1 *Programs*

CREDIT applications programs are developed under DOS 6800 System Software. However, they can be run only under TOSS System Software.
Under TOSS System Software only one application program can be held in memory. Hence, all the application processing for a PTS System is normally incorporated into one application program (which may, of course, be subdivided into modules).
When the total size of the TOSS-monitor plus the CREDIT application exceeds 64K bytes, different possibilities exist to run such applications on the PTS range of computers.

a) For systems having a maximum main memory capacity of 64K bytes, the only possibility is to use secondary memory (disk, flexible disk). From this secondary memory segments of the application are loaded into main memory at runtime, when necessary. This is under control of the memory management software.
b) For systems having an extended main memory, memory addressing upto 256K bytes, the whole application can be placed in main memory. Extended main memory may also be combined with use of secondary memory.
   A hardware feature, the memory management unit (MMU), enables memory addressing up to 256K bytes. This virtual storage technique is implemented, with using CREDIT memory management software

1.1.2.2 *Data Sets*

A data set is a reference to an input/output device or diskfile on which an application program may perform input/output operations. More than one data set may be configured in a single device. For example, a journal printer, tally roll printer and front feed printer are combined in the PTS 6221 Teller Terminal Printer. However separate input/output operations can be performed on each of the three data sets.

1.1.2.3 *Terminal Classes*

In a PTS System there is normally a device configuration at each of several work positions. Each device configuration comprises one or more devices. Some work positions may have the same type of device configuration, e.g. bank tellers would normally all use the same type of configuration. There are normally other work positions with different configurations. 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 types of transaction, identical program code is used to service each of these work positions.

1.1.2.4 *Tasks*

The CREDIT language enables the programmer to utilize the same set of CREDIT statements for each work position in a terminal class.
This is achieved in the following manner.
The interpretive object code generated from CREDIT programs is re-entrant. This means that a number of independent tasks can be achieved, all executing a single copy of the application program. Each time data is sent from a work position a task is activated by the TOSS Monitor. Thus, several tasks can be active at the same time for a number of terminal classes.
The TOSS Monitor schedules the various tasks so that, at any time, several tasks may be waiting for input/output to be completed, whilst other tasks are queued waiting for execution. Though only one task may be executed at a given instant, the overall impression is that all work positions are being serviced simultaneously.
Each task is assigned a unique task identifier by the system. This identifier is derived from the task identifier assigned to each terminal class by the programmer. With extended main memory, the TOSS-monitor always resides in the first 64Kbytes of main memory.

1.1.2.5 *Work Blocks*

One or more work blocks must be assigned by the programmer to each terminal class. These work blocks define areas of memory which may be used as working storage for e.g. input/output buffers. Dummy work blocks redefine these areas of memory. Swappable workblocks are stored on disk and will only on request be loaded into main memory.

1.1.3 *Program Design*

1.1.3.1 *General*

It is recommended that CREDIT programs be subdivided into modules. Each module should contain the statements necessary to perform a logically discrete processing step. There must be one main module in each program. This module will contain a complete data division headed by the DDIV directive. The remaining modules must not define a data division.

They should contain, instead, a DDUM directive followed immediately by the procedure division directive PDIV.
The result of this is that a single date division will be used by all modules in the program.
At least one terminal class should contain a program start point definition.
The remaining modules of the program may contain the statements required for the various types of transaction which the program is designed to process.
It is recommended that each module be devoted to the processing of a single transaction type.
It is the responsibility of the programmer to identify individual transaction types within a terminal class. This can be accomplished, for example, by testing a transaction code keyed-in at the work position by the user.
CREDIT programs may call subroutines written in PTS Assembler.
Certain system functions can be utilized only via Assembler programs. So it may be necessary to write a mixed CREDIT/Assembler program. However, the main module must always be written in CREDIT.

1.1.3.2 *Disk Resident Programs*

This way of extending the memory will lead to a decreasing of the performance, compared to memory extension with the memory management unit.

The code part consist of program segments just as for extended main memory. However, here the number of memory pages are not sufficient to permit all segments to be loaded in main memory together. The tasks have, as for other type of system resources to compete for main memory. The memory page replacing technique used is the least recently used method. This indicates that when the load in the computer goes down e.g. only a few tasks are running, these tasks will get a relatively large amount of main memory each. In situations of heavy computer load the tasks will get only the amount of main memory that is absolutely necessary. The dynamic allocation of main memory, when the system condition change, is controlled and supervised by the operating system itself. When looking at the code part it is important to consider the fact that the different tasks are using the same code to a great extent. In almost every application some or a lot of tasks are doing the same work on different physical work stations. These tasks are running the same instruction sequences but they are working on different and partly unique data areas. The situation above is valid for terminal systems in general. However the memory management technique is designed to handle also systems where the work within the system is delegated to a number of unique "specialist" tasks, each of them running its own program sequence. The difference will be that the competition for main memory will be harder in last-mentioned cases.

If no MMU is present the page size can be chosen to every value between sector size, 400 bytes, to 64K bytes, (also for flexible disk). The segmentation of the code part is made at linking time and the segments consists of; interpretable code, literal pool and address tables of the segment. Branches and subroutine calls will be solved automatically invisible to the user. Every segment can be loaded anywhere in main memory (in a page) and this decision is made by the system exclusively. Actually the only thing the user has to do is to define the program segment size.
The fetch policy used is, to load the segment at the point of time when it is needed, since it is very difficult to predict what segments will have to be loaded in a near future.

The replacement policy used e.g. the decision of which segment to overload when a new segment has to be loaded, is the Least Recently Used Method (LRU). A queue is built up telling which page in main memory to be replaced next time. This queue will be dynamically updated by the system each time a task is reactivated.

Note that the method described above implies that no dead-locks can appear, since there is always place for a new segment in main memory.

To take care of error situations (disk not operable, segment impossible to read) a special entry is defined in the resident part of the application program. (REENTER).

This virtual memory solution gives enough flexibility to the programmer to optimize the program execution. The most important thing is the concept of locality. When writing an application for a virtual memory system, the programmer should try to pack the frequently used modules to as small number of resulting segments as possible. In practice the following things can for example be considered when writing an application:
- remove exception and error-handling routines from the main path of the program.
- put all low Use routines in segments on their own.
- routines should be placed close to the routines they call or are called by.

Following rules should be noted, improving the locality of the program:
- there will be empty areas in the end of the pages, due to impossibility to make all the segments to the same size. The programmer, however, has the possibility to keep these empty areas at a low level.
- to have the possibility to build up and restructure the program segments, the programming technique to be used should be strongly modular.
- literals are placed in the segments where they are used.

1.1.3.3 *Extended Main Memory (up to 256K bytes)*

When using extended main memory, a special hardware feature the memory management unit (MMU) must be present.
The page size in systems with memory management unit (MMU) may be chosen by the user and should be a multiple of 1K bytes. This hardware feature allows a very fast paging system compared to disk as paging device. The page size is selected during linking. (TLK command, see chapter 2.3). The same rules are valid as mentioned for disc resident applications.

1.1.3.4 *Extended Main Memory and Disk Resident Programs*

The same rules are valid as mentioned for disk resident applications. The memory page replacement technique used, is the Least Recently Used Method, which will guarantee that the memory pages most frequently used will most of the time be situated in main memory.

1.1.4 *Source Input Format*

A CREDIT source program can be read into the PTS 6800 System using one of a variety of source input devices. Regardless of the input device used, the source data must have the following form.
A source line is an 80-character card image. If the input device allows records of variable length (console typewriter) each record must contain no more than one source statement. Input records longer than 80 characters are truncated, whereas shorter records are augmented by spaces up to column 80.

The source line is subdivided into four fields: label field, operation field, operand field and comments field. The label field begins in column 1. The label, operation and operand fields are each terminated by a tabulation character (\) or at least one space each. The operand field extends at maximum to column 71. If there are no non-space characters following the label (if any) before column 30, the rest of the statement is interpreted as a comment. Columns 73–80 are ignored in the translation process.
An asterisk in column 1 indicates that the source line is a comment. A source line containing spaces in columns 1–71 is ignored.
If column 72 contains a "C", the next line is interpreted as a continuation. For fixed length input records, the operand field may be terminated by a comma (leaving spaces up to column 72), the next operand starting on the continuation line. If a value inside quotes is split between two lines, all columns up to 72 are significant. For variable length records the operand field is terminated by two tabulation characters followed by a "C" for continuation. In this case, the character positions from the first tabulation character up to column 71 are not significant, and the operand field is immediately continued on the next line.
In continuation lines, the label and operation fields should be empty.

1.1.5 *CREDIT Syntax Definition*

The following symbols (Backus/Naur-Form) are used to define the syntax of CREDIT statements:

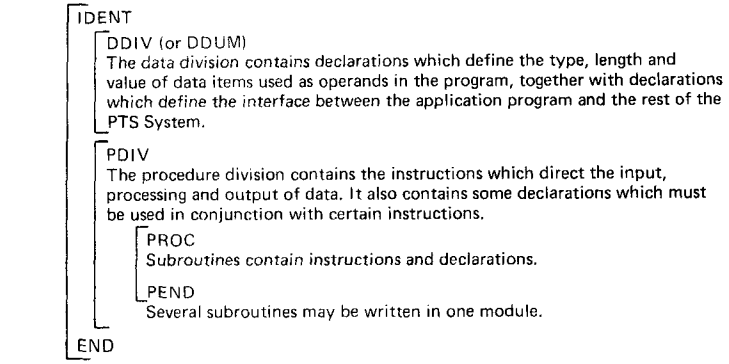
:: =	is defined as
␣	space
[ ]	the syntatic items between these square brackets may be omitted
{ }	select one of the items between the braces
a|b	select either a or b. This has the same meaning as braces. It is used with long strings.
...	ellipsis indicates that the last syntatic item may be repeated.

These symbols are used throughout the manual to define the syntax of CREDIT statements. They are also used in the parameter syntax definition in appendix 1.

1.2 Directives

1.2.1 *Structure Directives*

The framework of a CREDIT module is constructed from the directives IDENT, DDIV, DDUM, PDIV, PROC, PEND and END. The use of these directives is illustrated below:



The IDENT and END directives define the start and end of a module. They must be the first and last statements, respectively, of the module.
The DDIV or DDUM directive defines the start of the data division. It must be the second statement in the module. DDIV is used in the main module of a program. DDUM is used in all other modules.
The PDIV directive defines the start of the procedure division. The PROC and PEND directives define the start and end of a subroutine.
The IDENT, DDIV (or DDUM), PDIV and END directives must appear once only in each module. The PROC and PEND directives may be repeated. However, subroutines may not be nested. That is, two or more PROC directives cannot be written without an intervening PEND directive. Subroutines may, though, perform other subroutines.

1.2.2 *Linkage Directives*

CREDIT modules which have to be linked into an application program contain references to statements or subroutines in other modules. In order to achieve the correct inter-module linkages, entry points and external references must be specified. The ENTRY and EXT directives are used for this purpose. They must be written in the procedure division.
There must be at least one START directive for each program. When the TOSS System is started (i.e. the TOSS Monitor is loaded and the application program begins execution) a task is activated for each work position in the System. The tasks are activated at the start points specified in the START directives of the relevant terminal classes. The START directive(s) must be written in the data division and must be specified as entry points (ENTRY).

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 start points become pending. They will be activated after the first task has executed an EXIT instruction. When using memory management, the REENTER directive refers to a closing routine in case of a page fault or read error on disc. The statement identifier in this directive must be declared as ENTRY in the module it is defined.

1.2.3 *Listing Directives*

The directives LIST, NLIST and EJECT are used to control the CREDIT listing during the translation process. They may be written in any part of the module. The OPTNS directive must follow after the DDUM or DDIV directive.

1.2.4 *Equate directive*

This directive is used to equate an identifier with a value. When this identifier is used in an instruction the Translator automatically replaces it with the specified value, i.e. the instruction is translated as if the programmer had actually written the value in the instruction.
The directive may be used free in the procedure division (PDIV), but it should follow the ENTRY and EXT directives.

1.2.5 *Parameter directive*

The directives PFRMT, PKTAB and PLIT define the type of formal parameter, declared in the heading of the subroutine and must be used in two byte addressing mode or when a format table is used as formal parameter, or when in one byte addressing mode the literal constant name, keytable name or formatlist name, does not begin with a $ sign. These directives follow the PROC directive immediately.

1.2.6 *Directive reference*

This section describes the syntax and use of each directive. The possible values of the variables in the directives is given in appendix 1. The notation conventions are described in section 1.1.5.

DDIV *Data Division* DDIV

Syntax: ␣ DDIV ␣ [module-name]

Type: Structure directive

Description: Indicates the beginning of the data division of a module and causes a page feed in the listing during translation.

If module-name is specified, the data division statements will be fetched from the module indicated by module-name.

However, these data divisions will be entirely separate at execution time. The use of *common* data divisions is achieved through the DDUM directive.

No more than one module in a CREDIT program may contain a DDIV directive. The remaining modules must use the DDUM directive.

DDUM *Data division dummy* **DDUM**

Syntax: ␣ DDUM ␣[module-name]

Type: Structure directive.

Description: Beginning of a data division is indicated and the data division statements will be fetched from the module indicated by module-name.

No object code is output during processing of the data division.

Note: A DDUM-module may contain declaration of data items. The only difference between DDUM and DDIV directives is, when DDIV is declared the object modules of the data division are output on a /0 type file.

EJECT *Eject* **EJECT**

Syntax: ␣ EJECT ␣

Type: Listing directive.

Description: Listing will be continued at the top of the next page.

END *End* **END**

Syntax: ␣ END ␣

Type: Structure directive.

Description: This directive terminates a module. It must be the last statement appearing in each module.

ENTRY *Entry Point* **ENTRY**

Syntax: ␣ ENTRY ␣ { subroutine-identifier / statement-identifier }

Type: Linkage directive.

Description: The statement-identifier or subroutine-identifier within this module may be referred to by other modules.

Each identifier may comprise a maximum of six characters.

EQU *Equate* **EQU**

Syntax: equate-identifier ␣ EQU ␣ value-expression

Type: Equate directive.

Description: Equate-identifier takes on the type and value as specified by value-expression. The value must be between 0 and 255 inclusive. It should be used after the ENTRY and EXT directives, but may be written everywhere in the procedure division.

Example:

```
KEY     EQU     X'09'
KEY2    EQU     KEY+2
```

EXT — *External reference* — EXT

Syntax: ␣ EXT ␣ external-identifier

Type: Linkage directive.

Description: External-identifier points to a statement identifier or a subroutine identifier which is not in this module.

External-identifier consists of a maximum of six characters.

IDENT *Identification* **IDENT**

Syntax: ␣ IDENT ␣ module-name

Type: Structure directive.

Description: This directive specifies the name given to a module. It must always be present and must be the first statement in a module. Module-name may consist of a maximum of 8 characters, but it is truncated by the translator to 6 characters.
If comment follows the IDENT directive, a module name of at least 6 characters is recommended.

INCLUDE *Include* **INCLUDE**

Syntax: ␣ INCLUDE ␣ module-name [, LIST]

Type: Directive.

Description: This directive enables source code to be shared between modules in an application. The directive is used in the procedure division. The contents of the auxiliary file specified by module-name is included in the translation process. The auxiliary input file must not include the module directives IDENT, END or an INCLUDE directive.
If the option, LIST is specified, the source statements from the auxiliary file are listed. Otherwise they are not listed.
In the program listing, lines from the auxiliary file are identified by a hypen which follows directly the line number.

Example: ␣ INCLUDE MAUX, LIST

LIST

List

LIST

Syntax: ␣ LIST ␣

Type: Listing directive.

Description: CREDIT source code listing is resumed after it has been suspended by a NLIST directive.

NLIST *No list* **NLIST**

Syntax: ␣ NLIST ␣

Type: Listing directive.

Description: CREDIT source code listing is suspended from the point where this directive is given until either the END directive or LIST directive is met. Lines which contain syntax errors will continue to be printed during this phase.

OPTNS

Options

OPTNS

Syntax:

OPTNS LINES=value [,LITADR=decimal-integer] [,ADRMOD=decimal-integer]
LITADR=decimal-integer [,LINES=value] [,ADRMOD=decimal-integer]
ADRMOD=decimal-integer [,LINES=value] [,LITADR=decimal-integer]

Type:

Directive.

Description:

This directive should follow immediately after a DDIV or DDUM directive and is valid for the whole program (global). The keyword parameters are LINES, LITADR and ADRMOD. They have the following significance.

LINES — Value specifies the number of lines per page on the output listing.

LITADR — Literal constant, pictures, keytables and formats are addressed by a one or two byte addressing system. A four digit, decimal integer indicates which addressing system is used. The first digit refers to literal constant, the second digit to keytables, the third to pictures and the fourth to formats. Each digit may have a value 1 or 2.

First digit:
- 1 One byte addressing system selected for literal constants.
- 2 Two byte addressing system selected for literal constants.

Second digit:
- 1 One byte addressing system selected for keytables.
- 2 Two byts addressing system selected for keytables.

Third digit:
- 1 One byte addressing system selected for pictures.
- 2 Two byte addressing system selected for pictures.

Fourth digit:
- 1 One byte addressing system selected for formats.
- 2 Two byte addressing system selected for formats.

Default value LITADR=1111.

OPTNS *Continued* OPTNS

ADRMOD With one digit, a two byte addressing mode can be selected for data-items, data-sets, literal constants, keytables, formats and pictures.

1 One byte addressing for data-items, data-sets, literal constants, keytables, formats and pictures. With LITADR two byte addressing can be selected for literal constants, keytables, pictures and/or formats.

2 Two byte addressing for data-items, data-sets, literal constants, keytables, formats and pictures. LITADR is automatically set to two byte addressing.

Default value ADRMOD=1.

PDIV *Procedure division* **PDIV**

Syntax : ␣ PDIV ␣

Type : Structure directive

Description : Indicates the beginning of the procedure division and causes the data division processing to be terminated. A new page of program listing is thrown.

PEND *Subroutine end* **PEND**

Syntax: ␣ PEND ␣

Type: Structure directive.

Description: This directive marks the end of a subroutine. Formal parameters may not be referred to by an instruction which is written after a subroutine PEND directive.

PFRMT *Formal format list parameters* **PFRMT**

Syntax: — PFRMT — { identifier / $identifier }

Type: Parameter directive.

Description: In two-byte addressing mode this directive defines a formal parameter of the type format list reference. It follows the PROC directive and must also be used when the formal parameter corresponds to a format table reference (FTABLE).

The $ sign is counted in the total number of characters in the identifier. (formal parameter) When the formal parameter does not start with a $ sign and the referenced type is format list, then the type has to be defined by a PFRMT directive, also in one byte addressing mode.

Example:

```
SUBF_PROC_$FORM1          (ADRMOD=2)
      PFRMT $FORM1
        .
        .
        .
SUBF_PROC_$FORM1          (ADRMOD=1)
        .
        .
        .
SUBF_PROC_FORM1           (ADRMOD=1)
      PFRMT FORM1
        .
        .
        .
```

PKTAB *Format key table parameter* **PKTAB**

Syntax: ⎵ PKTAB ⎵ { identifier / $identifier }

Type: Parameter directive

Description: In two-byte addressing mode, this directive defines a formal parameter of the type key table reference. It follows the PROC directive.

The $ sign is counted in the total number of characters in the identifier (formal parameter). When the formal parameter does not start with a $ sign and the referenced type is key table, then the type has to be defined by a PKTAB directive, also in one byte addressing mode.

Example:

```
SUBK ⎵ PROC ⎵ $SKTB1          (ADRMOD=2)
       PKTAB   $KTB1
         .
         .
         .
SUBK ⎵ PROC ⎵ $KTB1           (ADRMOD=1)
         .
         .
         .
SUBK ⎵ PROC ⎵ KTB1            (ADRMOD=1)
       PKTAB   KTB1
         .
         .
         .
```

PLIT *Formal literal parameter* **PLIT**

Syntax: ␣ PLIT ␣ {identifier / $identifier}

Type: Parameter directive

Description: In two byte addressing mode, this directive defines a formal parameter of the type literal reference. It follows the PROC directive.

The $ sign is counted in the total number of characters in the identifier. (formal parameter) When the formal parameter does not start with a $ sign and the referenced type is literal constant then the type has to be defined by a PLIT directive, also in one byte addressing mode.

Example:

```
SUBL ␣ PROC ␣ $LITC          (ADRMOD=2)
       PLIT   $LITC
        .
        .
SUBL ␣ PROC ␣ $LITC          (ADRMOD=1)
        .
        .
        .
SUBL ␣ PROC ␣ LITC           (ADRMOD=1)
       PLIT   LITC
        .
        .
        .
```

PROC *Subroutine start* **PROC**

Syntax: Subroutine-identifier␣PROC␣[formal-parameter] [,formal-parameter] ....

Type: Structure directive.

Description: The PROC directive forms the heading of a subroutine. A maximum of eight formal parameters may be specified. (ADRMOD=2).
With ADRMOD=1 the number of formal parameters is limited to 8 bytes. When using literal constants, keytables or format lists as formal parameter in two byte addressing mode, selected with the LITADR option, the maximum number of formal parameters is decreased.

REENTER — *Reenter* — REENTER

Syntax: — REENTER — statement-identifier.

Type: Linkage directive.

Description: When using memory management, this directive refers by means of the statement identifier to a closing routine. This routine, written by the user, will be executed when it is impossible to read a code segment in main memory or from disk.
This directive should be located following the start directive. Statement-identifier must be declared as ENTRY in the module it is defined.

START — *Start Point* — START

Syntax: ␣ START ␣ { statement-identifier / external-identifier }

Type: Linkage directive.

Description: For each terminal class, START indicates the first instruction to be executed in the program.
This directive should precede the STACK declaration.
The start address has to be specified as an ENTRY in the relevant module.

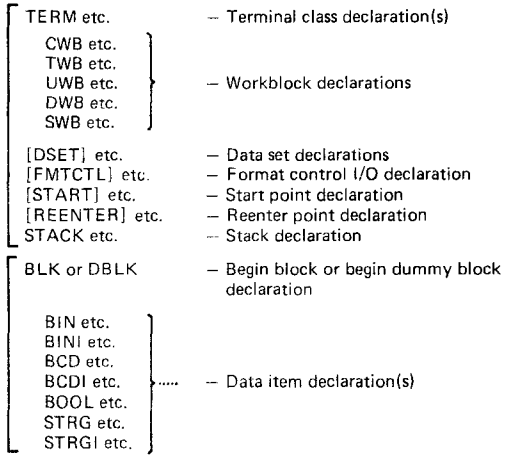
1.3 Data Division

1.3.1 *Introduction*

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 application program and the rest of the PTS System. The use of directives in the data division is discussed in Section 1.2. The general layout of the data division is shown below.

There must be one terminal class declaration for each terminal class used by the program. The terminal class declarations may be made in any sequence, but they must all appear at the start of the data division.

There must be one begin block declaration for each block identifier specified in the work block declarations. The begin block declarations may be made in any sequence and must appear after the last terminal class declaration.



1.3.2 *Terminal Class Declaration*

The TERM declaration identifies the terminal class with a unique two character task identifier. It is followed immediately by the relevant work block declaration(s), start point directive, data set declaration(s) and stack declaration.

1.3.3 *Work Block Declarations*

There may be a maximum of 15 work block declarations in each terminal class. These declarations refer to the block identifiers specified in the begin block declarations (BLK or DBLK).

Several work block declarations may refer to a single block identifier. This implies that these work blocks are defined under a single begin block declaration. For terminal work blocks, user work blocks and swappable work blocks this facility is merely a form of programming convenience. It does not mean that these work blocks are "overlayed" or "common". However, the facility saves the effort of writing out identical sets of declarations a number of times. For common work blocks, more than one reference to a single block identifier results in the generation of a single work block which can be used by several tasks.
Dummy work blocks are used to redefine a terminal-, common-, user-, or swappable work block in the same terminal class.
If a block identifier is used in more than one work block declaration, each declaration must always occupy the same position relative to the other work block declarations in the same terminal class. See the Declaration Reference Section (1.3.9) for an illustration of this rule.

1.3.4 *DSET, FMTCTL, START, REENTER and STACK*

The DSET declaration specifies those data sets which are to be used in this terminal class. The declaration is optional.
The FMTCTL declaration indicates the format control I/O feature will be used in this terminal class.
The START directive indicates the start point for the task(s) associated with this terminal class.
The REENTER directive can be used with memory management, to refer to a closing routine which will be executed when it is impossible to read a code segment in main memory or from disk.
The STACK declaration specifies the size of the stack to be allocated to this terminal class. The declaration is optional.

1.3.5 *Begin Block Declaration*

1.3.5.1 *General*

The BLK or DBLK declaration identifies the work block defined by the data item and array declarations which follow.
Work blocks are used to define areas of memory which may be used for input/output buffers and/or work locations. Dummy work blocks do not occupy memory locations but are redefining existing work blocks in the same terminal class.
The instruction of a CREDIT program are held in as a single memory copy during execution. However, the number of copies of work blocks which exist in memory depends upon the configuration of the PTS System used by the application. The number of work blocks needed depends upon the number of work positions and upon the number of users who will operate these work positions. The work blocks are actually generated when the application program is loaded into core.
They are generated by a part of TOSS System Software, the SYStem LOADing program (SYSLOD). The use of SYSLOD is discussed in part two of this manual.

1.3.5.1 *Terminal Work Blocks*

One or more terminal work blocks (TWB) may be defined for each terminal class within a program. A separate copy of the terminal work blocks will exist in memory for each task activated for a particular terminal class. Each copy may be used only by its associated task and cannot be used for exchanging data with other tasks.

1.3.5.3

... may be defined for each terminal class with a ... work blocks is generated. They may be referenced by ... the task's terminal class. If the same block identifier is used in the work block ... terminal class, then the tasks belonging to these ... the same common work block.

1.3.5.4

In this ... operator. It may be necessary for the ... information, e.g. cash accumulators, for each user. ... one-to-one relationship between work positions and ... number of users as work positions. The users may rotate ...
To ... areas of memory are required which are associated with individual ... positions. These areas of memory are called user work ...
One ... with different block identifier, may be defined for each ...
Tasks ... work block if they belong to a terminal class which ... user work blocks. Furthermore, a task may only refer to a user work ... USE instruction specifying the block identifier and index identifier ... An index identifier is a reference to a decimal integer from 1 ... which is used to differentiate between user work blocks of the same ... work block type may be connected to one task. It is possible ... the same user workblock type at the same time. This ... with care. It is the responsibility of the application ... which may result from this type of access.

1.3.5.5

A dummy ... work block referenced by the block identifier, between the ... declaration. Definition starts at the beginning of the referenced ... terminal class, and continues sequentially. Redefinition ... type of data items but is restricted to the size of the defined ...

Example:

...

...

...

...

...

...

...

DB1	DBLK		
DU1	BCD	8	redefines BN1 and BN2
STRU1	STRG	11	redefines D1, D2 and ST1

When data item DU1 is processed the memory locations for BN1 and BN2 are used as decimal-data-item. The user must be aware of this situation when data-item BN1 or BN2 is processed afterwards, while it is to be processed as binary data item. Dummy work blocks may contain data-items of type BOOL, BIN, BCD, STRG, BINI, BCDI or STRGI. No initial values may be assigned to these data-items as no memory space is allocated. For data-items of type decimal and string, the length of each item must be declared explicitly.

1.3.5.6 *Swappable Work Blocks (SWB)*

Swappable work blocks are disk resident and in use similar to user work blocks. A swappable work block type, different block-identifier, is read into memory only when it is attached to a task with the USE instruction.
One copy of a swappable work block type may be present in main memory at the same time and is allocated to one task as long as it stays in main memory. However, they can be used by a multiple of tasks but not simultaneously. One area in main memory is reserved for each swappable work block type per task.
The swappable work block is set under exclusive access on the diskfile $SWAP. Exclusive access is released when the instruction UNUSE is executed for that particular work block type. Data which is not frequently used can be stored in these work blocks. A swappable work block is rewritten on disk after it is detached from the task, by executing the UNUSE instruction. The work blocks are copies onto a disk file $SWAP, at system loading time by the system loading program (SYSLOD).
File $SWAP must be a standard file (S type) with record length 400 and blocking factor 1.

1.3.6 *Data Item and Array Declarations*

1.3.6.1 *General*

These declarations describe the data items and arrays of which the work blocks are composed.

Data items may be binary coded decimal (BCD), binary (BIN), boolean (BOOL) or string (STRG). The mnemonics BCD, BIN, BOOL and STRG are known as "data item types". Arrays may be binary coded decimal (BCDI), binary (BINI) or String (STRGI). The mnemonics BCDI, BINI and STRGI are known as "array types". These types should be distinguished from the "value types" described below.

Any BIN or BINI declarations must appear at the start of the work block.

1.3.6.2 *Data Item Declarations*

A data item declaration has the following general format:

identifier␣ { BCD / STRG } ␣ data-item-specification

identifier␣ BIN ␣ [data-item-specification]

identifier ␣ BOOL ␣ [[TRUE / T / FALSE / F]]

Data-item-specification specifies the length, value type and value of the data item. It has the following general format in both data item declarations and array declarations:

```
{length [ [value-type] ['value'] ]
 value-type ['value']
 'value'                           }
```

where:

Length is a decimal integer indicating the length of the item. It has the following significance:

BCD and BCDI — Length indicates the number of (4 bit) digits in the data item.

BIN and BINI — Length need not be used. If value-type is W, length may only be 1. If value-type is X or D, length indicates the number of (4 bit) digits in the data item. If value-type is C, length indicates the number of (8 bit) ISO-7 characters in the data item.

Value-type is one of the characters D, W, X or C, and specifies whether value is to be stored in memory as decimal (D), binary (W), hexadecimal (X) or string (C). The following combinations of data item type and value-type are allowed:

BCD and BCDI	—	D or X
BIN and BINI	—	C, D, W or X
STRG and STRGI	—	C or X

If value-type is not specified, the following defaults are assumed:

BCD and BCDI	—	D
BIN and BINI	—	W
STRG and STRGI	—	C

Value specifies the value to be assigned to the data item. If value is longer than the specified length will allow, then:
BCD, BCDI, BIN and BINI — value is truncated at the left. The least significant digits are placed in the data item.
STRG and STRGI — value is truncated at the right. The leftmost characters in value are placed in the data item.
If value is shorter than the specified length will allow, then:
BCD, BCDI, BIN and BINI — value is right adjusted and zero filled on the left.
STRG and STRGI — value is left adjusted and the data item is filled on the right by repeating the rightmost character of value.
If value is not specified, then:
BCD, BCDI, BIN and BINI — value is assumed to be zero.
STRG and STRGI — value is assumed to be ISO-7 spaces.
Value must be written as a decimal number, a hexadecimal integer or a character string, depending upon value-type. The rules are as follows:

D — decimal number
X — hexadecimal integer
W — decimal number
C — character string

Value will be stored in the data item in the following way:
BCD and BCDI – The digits in value are placed directly in the data item without conversion. Each digit occupies four bits.
BIN and BINI – If value-type is W, value is converted from decimal to binary representation and then stored. If value-type is X or D, value is placed in the data item without conversion. Each digit occupies four bits. If value-type is C, the ISO-7 hexadecimal representation of the characters is placed in the data item. One character occupies eight bits
STRG and STRGI – If value-type is C, the ISO-7 hexadecimal representation of the character is placed in the data item. One character occupies eight bits. If value-type is X, value is placed directly in data item without conversion. Each digit occupies four bits.

If length is omitted, the length of the data item is implied by the specified value-type and value.

Binary (BIN) data items have a fixed length of one word. For this type of data item length, value-type and value may all be omitted.

1.3.6.3 *Array Declarations*

An array declaration has the following general formats.

array-identifier ␣ { BCDI / STRGI } ␣ (dimension [,dimension]), data-item-specification [, 'value'] ....

array-identifier ␣ BINI ␣ (dimension [,dimension]) [,data-item-specification] [,'value'] ....

The use of the data-item specification has been described above. If there are fewer "values" in data-item-specification than there are data item occurrences in the array, the last "value" specified is repeated in the remaining data item occurrences.

If a value is defined, longer than the value defined by the first list element, it is truncated according to the rules defined in 1.3.6.2.

"dimension" (or the product of the dimensions if there are two) indicates the number of occurrences of the data item in the array. Each dimension must be a decimal integer. A maximum of two dimensions is allowed.

When setting an index to refer to an array, the first occurrence in each dimension is always counted as one. For a two dimensional array both indices must lie in the range 1 to 255. A one dimensional array is not restricted in this way.

An initial value may be assigned to an array. This is done by listing, in the array declaration, the values to be assigned to each occurrence. Each value must appear in quotes and must be separated from the next value by a comma. For example:

TAB␣BINI(4,4), '1', '2', '3', '4', '5', '6', '7', '8', '9', X'A', X'B', X'C', X'D', X'E', X'F', X'0'

The Translator will set these values up in the array row by row from left to right, as follows:

0001	0002	0003	0004
0005	0006	0007	0008
0009	000A	000B	000C
000D	000E	000F	0000

The array will be referenced in the following manner:

TAB (X, Y) has the value 2 (where contents of binary-data-items, X = 1 and Y = 2)

TAB (X, Y) has the value X 'A' (where contents of binary-data-items, X = 3 and Y = 2)

1.3.7 *Data Items*

A data item is the most elementary unit of data that can be used as an operand in a CREDIT instruction. Data items are defined by data item or array declarations and are refered to by their data item identifier or array identifier and dimension(s):

Type of declaration	Type of reference
data-item-declaration	data-item-identifier
array-declaration	array-identifier (index-identifier-1 [,index-identifier-2])

In this Manual data items are referred to as "decimal data items", "string data items" etc. **These data items may be defined by *either* data items *or* array declarations (except for boolean):**

Type of declaration	Type of data item
binary-data-item-declaration binary-array-declaration	binary-data-item
boolean-data-item-declaration	boolean-data-item
decimal-data-item-declaration decimal-array-declaration	decimal-data-item
string-data-item-declaration string-array-declaration	string-data-item

Data items which have a special significance in a group of instructions have been given special names. There are three data items in this category: index, pointer and size. These names are used in order to simplify the descriptions of these groups of instructions. However, the data items are held in memory and operated upon in exactly the same way as other data items.

An index is used in the IB and PERFI instructions.
It is used in these instructions to select a particular statement identifier or external identifier from a specified identifier list. Index must always be defined as a binary data item.

A pointer is used in the COPY, DLETE, EDIT, INSRT, MATCH and XCOPY instructions. It is used in these instructions to point to an individual character in a data item. Pointer must always be defined as a binary data item.

A size is used in the COPY, DLETE, INSRT, KI, MATCH, NKI, WRITE, IREAD, IWRITE, RREAD, RWRITE and READ instructions. It is used in these instructions to indicate the size of the data item segment upon which the instruction is operating. Size must always be defined as a binary data item.

1.3.8 *Work Blocks*

For each work block the translator generates a 16 or 256 entry pointer table maximum, depending on the value assigned to ADRMOD in the OPTNS directive. The entries in each of these tables point to the non-boolean (i.e. binary, decimal or string) data items and arrays in the associated work block. Each non-boolean data item in a work block occupies one entry in the pointer table. Each array occupies two entries. The number of non-boolean data items and array declarations in each work block must be such that the number of entries in the pointer table does not exceed 16 or 256.

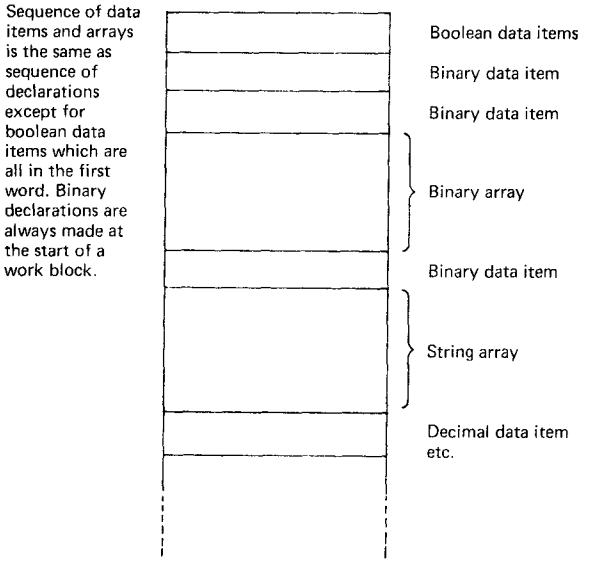
The following combination is possible (ADRMOD=1):

4 data item declarations	— requiring	4 entries
6 array declarations	— requiring	12 entries
Total		16 entries

The pointer table is not used for boolean data items. This is because all boolean data items in a work block are stored in the first word of that block. The maximum number of boolean data items in a work block is therefore 16.

Memory for non-boolean data items and arrays is allocated in the same sequence as the corresponding data item and array declarations appear in the CREDIT listing.

An example work block layout is shown below:



The last data item declared in a work block may not have a start from a displacement higher than 32767.

1.3.9 *Declaration Reference*

This section describes the syntax and use of each declaration. The possible values of the variables in declarations is given in appendix 1. The notation conventions are described in section 1.1.5.

BCD *Decimal data item* **BCD**

Syntax: data-item-identifier ␣ BCD ␣ data-item-specification

Description: Decimal data items are of variable length, varying from 1 to 510 decimal digits (sign included) and occupying a whole number of bytes.

Example:

			memory representation:
R	BCD	4'–9'	X'D009'
S	BCD	X'BFF40'	X'BFF400'

The sign is the most significant tetrad: D = negative, B = positive

The character 'F' in the above example denotes a "BCD space". This character is treated as zero in arithmetic operations. However, if the data item is edited and printed, BCD spaces will appear as blanks. Under certain circumstances BCD spaces are generated by the MOVE instruction.

BCDI *Decimal array* **BCDI**

Syntax: array-identifier ␣BCDI␣ (dimension [,dimension]),
data-item-specification ['value'] ...

Description: Within the array all elements have a fixed size. The size for decimal array elements may vary from 1 to 510 digits (sign included). The size of an element is derived from the first element in the list. A one dimensional array may contain maximum 32767 elements. A two dimensional array may vary from 1 to 255 elements per row and 1 to 255 elements per column.
(Maximum 255*225 elements)
When an array is partly initialized, the last defined element will be copied until all remaining elements are filled.

Example:

```
TAB1    BCDI    (25),6D'-94278'
TAB2    BCDI    (3),'1','2','3'
TAB3    BCDI    (2,5),6D'0'
```

BIN *Binary data item* **BIN**

Syntax: data-item-identifier ␣ BIN ␣ [data-item-specification]

Description: Binary data items are allocated full words and thus have a fixed length of 16 bits. All binary data items within a block must be declared before the decimal and string data items and arrays.
The data item value must be within the range −32768 to 32767.

Example:

			memory representation:
A	BIN	W'20'	X '0014'
B	BIN	X'FF'	X '00FF'
C	BIN	3D'100'	X 'B100'
D	BIN	2C'NO	X '4E4F'
E	BIN		X '0000'

BINI *Binary array* **BINI**

Syntax: array-identifier ␣BINI ␣ (dimension [,dimension])
[,data-item-specification] [,'value'] ...

Description: Binary array elements are allocated full words and each element has a fixed length of 16 bits.
The value of an element in the array must be within the range –32768 to 32767.
A one dimensional array may contain maximum 32767 elements. A two dimensional array can vary from 1 to 255 elements per row and 1 to 255 elements per column. (Maximum 255*255 elements). When an array is partly initialized, the last defined element will be copied until all remaining elements are filled.

Example:
```
TAB1    BINI    (100),'0'
TAB2    BINI    (5), '1', '2', '3', '4', '5'
```

BLK *Begin block* **BLK**

Syntax: block-identifier ␣ BLK

Description: Defines the beginning of a work block.
The three leading characters of the block-identifier should be unique since the identifier is truncated if longer. The block-identifier has to correspond with the one specified in the work block directive (TWB, CWB, UWB or SWB).
The begin block declaration must be followed by at least one data item declaration or array declaration.

BOOL *Boolean data item* **BOOL**

Syntax:

data-item-identifier ␣ BOOL ␣ [{ TRUE / T / FALSE / F }]

Description: The length of a boolean variable is always one bit. A value may be declared by selecting one of the following:-
TRUE, T, FALSE or F.
TRUE or T corresponds with "1" and FALSE or F corresponds with "0". The default value is always FALSE (zero).

Boolean variables may not be indexed.

Example:

```
FLAG        BOOL        TRUE
```

CWB *Common work block* **CWB**

Syntax: ␣ CWB ␣ block-identifier

Description: The block-identifier refers to a begin block declaration (BLK). The three leading characters of the block identifier must be unique. The identifier is truncated if longer.
The same block-identifier may be used in different terminal classes. If this is done the common work block declaration must always occupy the same position relative to the other work blocks in the same terminal class.
A common work block is only common for those terminal classes in which it is defined.

Example:

Legal:

```
TERM A0
CWB A ◄┐
TWB B  │
UWB C  │
TERM B0│
CWB A ◄┘
TWB D
UWB E
```

Illegal:

```
TERM A0
CWB A ◄┐
TWB B  │
UWB C  │
TERM B0│
TWB D  │
UWB E  │
CWB A ◄┘
```

DBLK *Begin dummy block* **DBLK**

Syntax: Block-identifier ␣ DBLK

Description: Defines the beginning of a dummy work block. The three leading characters of the block-identifier should be unique since the identifier is truncated if longer. The block-identifier has to correspond with the one defined in the DWB directive. (block-identifier-1).
This begin dummy block declaration must be followed by at least one data item declaration or array declaration. It serves to declare a redefinition of a work block.

DSET *Data Set* **DSET**

Syntax: data-set-identifier ␣ DSET ␣ FC = file-code
[, BUFL = decimal-integer]
[, DEV = device-type]
[, BUFDS = data-set-identifier]

Description: A data set is an I/O device. Particular data sets may only be accessed from a task if the task belongs to a terminal class containing a DSET declaration for that data set.
The same dataset declaration may be used in different terminal classes. If this is done, the dataset declaration must always occupy the same position relative to the other dataset declarations in all terminal classes in which it is used.
The keyword parameters FC, BUFL and DEV may be written in any sequence. They have the following significance:

BUFL	Buffer length. If this parameter is present a fixed buffer is allocated. The parameter value is a decimal integer giving the length in characters.
BUFDS	Buffer data set. If this parameter is present, the preceding buffer length (BUFL) will be shared with the data set buffer indicated by the data-set-idientifier after BUFDS. The buffer size must be smaller than the shared buffer.
DEV	Device type. Parameter value consists of two letters (see following device type list).
FC	TOSS file code. Parameter value consists of two hexadecimal digits (see following file code list).

Any unrecognized keywords are ignored.

Example:

```
PRT     DSET    FC = 40, DEV = LP, BUFL = 120
OWNER   DSET    FC = 32, DEV = TV, BUFL = 100
SHARE   DSET    FC = 31, DEV = TR, BUFL = 36,
                                   BUFDS = OWNER
```

TOSS Device type:	Meaning:
CR	Card Reader
DC	Data Communication
DG	Graphic display
DI	Indicator display
DL	Logical disk file
DN	Numeric display
DY	Alphanumeric display (VDU)
GP	General printer

DSET *Continued* DSET

TOSS Device type:	Meaning:
II	Intertask input
IO	Intertask output
KA	Alphanumeric keyboard
KI	Keyboard indicators
KN	Numeric keyboard
LP	Line Printer
MT	Magnetic Tape (1/2 inch)
SI	System Operator's panel Switches
SO	System Operator's panel lamps
TK	Cassette Tape
TJ	Teller terminal printer : Journal print station
TR	Teller terminal printer : Tally roll print station
TV	Teller terminal printer : Voucher print station
TW	Typewriter

Recommended TOSS File codes : (hexadecimal)	Meaning:
10	System operator panel-in.
11	System operator panel-out.
12	Cassette recorder nr. 1
13	Cassette recorder nr. 2
15	Remote line test
20	Keyboard
25	Reserved for future use.
30	General Printer.
30, 31, 32	Teller printer (TJ, TV, TR)
40	Signal display
41	Numeric display
50	Character display
60	Data communication
70	Magnetic tape
80	Line printer
90	Card reader
A0, B0–B3, B6	Reserved for future use
C0–CF	Data management disk files
D0	Inter task communication-input
D1	Inter task communication-output
F0–FB	Reserved for system
FC–FF	Reserved for user

DWB — *Dummy work block* — DWB

Syntax: ␣ DWB ␣ block-identifier-1(block-identifier-2)

Description: Block-identifier-1 refers to the begin dummy block declaration (DBLK), where block-identifier-2 refers to a work block declared in the same terminal class, which will be redefined.
The three leading characters of both block identifiers must be unique. The identifiers are truncated if longer. The same block-identifier-1 may be used in different terminal classes. If this is done the dummy work block declaration must always occupy the same position relative to the other work blocks in the same terminal class.

Example 1:

```
        TWB     TB1
        DWB     DB1 (TB1)
          .
          .
TB1     BLK
          .
          .
DB1     DBLK
          .
          .
```

Example 2:

Legal

```
TERM    A0
TWB     TB1
TWB     TB2
DWB     DB2 (TB1)  <-+
  .                  |
  .                  |
TERM    B0           |
TWB     TB1          |
TWB     TB4          |
DWB     DB2 (TB1)  <-+
```

Illegal

```
TERM    A0
TWB     TB1
TWB     TB2
DWB     DB2 (TB1)
  .
  .
TERM    B0
TWB     TB1
DWB     DB2 (TB1)
TWB     TB3
```

FMTCTL *Format control I/O* **FMTCTL**

Syntax:

␣ FMTCTL␣ { INDS=data-set-identifier, OUTDS=data-set-identifier } / { OUTDS=data-set-identifier, INDS=data-set-identifier }

Description:

A format control I/O declaration, must follow the data set declarations immediately.
It specifies that this terminal class will use the format control I/O feature.
Data-set-identifier after INDS specifies the input device and after OUTDS the output device, used for format controlled I/O, FMTCTL and referenced data sets must be in the same terminal class.

Example:

```
DSKB    DSET    FC = 20, DEV = KB
DSDY    DSET    FC = 50, DEV = DY, BUFL = 90
DSGP    DSET    FC = 30, DEV = GP, BUFL = 90
        FMTCTL  INDS = DSKB, OUTDS = DSDY
```

STACK *Stack* **STACK**

Syntax: ␣ STACK ␣ size

Description: Allocates a user memory stack.
Size is a decimal integer indicating the size of the stack in bytes. If the stack declaration is omitted a default stack size of 128 bytes is allocated. This declaration should occur after the START or REENTER declaration.

STRG *String data item* **STRG**

Syntax: data-item-identifier ␣ STRG ␣ data-item-specification

Description: Strings are of variable length and occupy a whole number of characters. The length varies from 1 to 4095 characters.

Example:

			memory representation:
ALL	STRG	7'HEADING'	X'48454144494E47'
BTA	STRG	8X'42455441'	X'42455441'
BET	STRG	4'BETA'	X'42455441'
F1	STRG	5C'AB'	X'4142424242'
F2	STRG	5C'ABCDEF'	X'4142434445'

STRGI *String array* **STRGI**

Syntax: array-identifier ␣ STRGI ␣ (dimension [,dimension])
,data-item-specification ['value'] ....

Description: Within an array all elements have a fixed size. The size for a string array element may vary from 1 to 4095 bytes. The size of an element is derived from the first element in the list. A one dimensional array may contain maximum 32767 elements. A two dimensional array may vary from 1 to 255 elements per row and 1 to 255 elements per column. (Maximum 255*255 elements). When an array is partly initialized, the last element will be copied until all remaining elements are filled.

Example:

```
TAB1    STRGI    (10, 10), 32C' '
TAB2    STRGI    (10).40C
```

SWB *Swappable Workblock* **SWB**

Syntac: ␣ SWB ␣ block-identifier.

Description: The block-identifier refers to a begin block declaration (BLK). The three leading characters of the block-identifier must be unique. The identifier is truncated if longer.
The same block-identifier may be used in different terminal classes. If this is done the swappable workblock declaration must always occupy the same position relative to the other workblocks in the same terminal class.
A task can only access a swappable work block after the USE instruction has been executed by that task. It can be detached from the task by executing the UNUSE instruction.

Example:

Legal:

```
TERM A0
CWB  A
TWB  B
UWB  C
SWB  D ◄──┐
                │
TERM B0         │
CWB  E          │
TWB  F          │
TWB  G          │
SWB  D ◄────────┘
```

Illegal:

```
TERM A0
CWB  A
TWB  B
UWB  C
SWB  D ◄──┐
                │
TERM B0         │
CWB  E          │
TWB  F          │
SWB  D ◄────────┘
TWB  H
```

TERM *Terminal class* **TERM**

Syntax: ␣ TERM ␣ task-identifier

Description: This declaration followed by a maximum of 15 work block declarations describes a class of terminal.
The task identifier must consist of a letter followed by a digit or a letter.

Note: Task identifier is the same as the task identifier specified at system loading time. (SYSLOD)

TWB *Terminal work block* **TWB**

Syntax: ␣ TWB ␣ block-identifier

Description: The block-identifier refers to a begin block declaration (BLK). The three leading characters of the block identifier must be unique. The identifier is truncated if longer.
The same block-identifier may be used in different terminal classes. If this is done the terminal work block declaration must always occupy the same position relative to the other work blocks in the same terminal class.

Example: Legal:

```
TERM A0
CWB A
TWB B ◄┐
UWB C  │
       │
TERM B0│
CWB D  │
TWB B ◄┘
UWB E
```

Illegal:

```
TERM A0
CWB A
TWB B ◄┐
UWB C  │
       │
TERM B0│
CWB D  │
UWB E  │
TWB B ◄┘
```

UWB *User work block* **UWB**

Syntax: ␣ UWB ␣ block-identifier

Description: The block-identifier refers to a begin block declaration (BLK). The three leading characters of the block identifier must be unique. The identifier is truncated if longer.
The same block-identifier may be used in different terminal classes. If this is done the user work block declaration must always occupy the same position relative to the other work blocks in the same terminal class.
A task may only access a user work block after a USE instruction has been executed by that task. It can be detached from the current task by executing the UNUSE instruction.

Example:

Legal:

```
TERM A0
CWB A
TWB B
UWB C ◄──┐
                │
TERM B0         │
CWB D           │
TWB E           │
UWB C ◄──┘
```

Illegal:

```
TERM A0
CWB A
TWB B
UWB C ◄──┐
                │
TERM B0         │
CWB D           │
UWB C ◄──┘
TWB E
```

1.4 Procedure Division

1.4.1 *Introduction*

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

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

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

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

1.4.2 *Instructions*

1.4.2.1 *General*

The general format of an instruction is:

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

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

1.4.2.2 *Arithmetic Instructions*

These instructions are:

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

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

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

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

1.4.2.3 *Branch Instructions*

These instructions are:

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

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

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

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

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

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

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

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

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

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

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

1.4.2.4 *Input/Output Instructions*

These instructions are:

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

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

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

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

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

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

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

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

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

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

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

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

1.4.2.5 *Logical Instructions*

These instructions are:

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

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

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

1.4.2.6 *Scheduling Instructions*

These instructions are:

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

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

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

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

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

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

1.4.2.7 *Storage control instruction*

These instructions are:

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

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

1.4.2.8 *String Instructions*

These instructions are:

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

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

String instructions occupy from three to seven bytes of core.

1.4.2.9 *Subroutine Control Instructions*

These instructions are:

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

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

1.4.2.10 *Format control I/O instructions*

These instructions are shown in the table below?

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

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

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

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

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

1.4.3 *Declarations*

1.4.3.1 *Format lists*

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

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

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

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

An example of a format list is shown below:

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

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

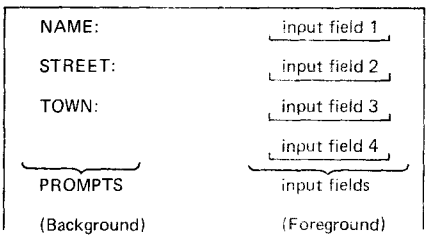
␣ ␣ TERMINALNN etc

where NN is the value from the data item TERM.

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

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

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



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

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

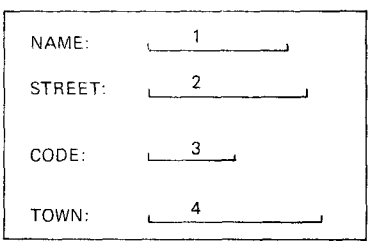
Two types of input fields are defined:

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

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

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

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

FINP .....   (3)

FKI  .....   (4)

FMEND
```



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

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

Sequence numbering of FKI-input fields only:

```
FRMT
 :
FKI  .....  (1)
 :
FKI  .....  (2)
 :
FINP .....
FKI  .....  (3)
 :
FMEND
```

```
NAME:    1
STREET:  2
CODE:
TOWN:    3
```

Sequence numbering for FINP-input fields only:

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

```
NAME:
STREET:
CODE:    1
TOWN:
```

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

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

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

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

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

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

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

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

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

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

Example: MOVE L_ : FMTITEM, SPINPUT

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

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

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

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

1.4.3.2 *Key Table Declaration*

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

1.4.3.3 *Parameter declaration*

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

1.4.4 *Subroutine handling*

1.4.4.1 *CREDIT subroutines*

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

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

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

Example:

```
        OPTNS     LITADR = 1111
SUB1    PROC      FORM1, LITC, KTB1, DAT1
        PKTAB     KTB1
        PFRMT     FORM1
        PLIT      LITC

        PEND
SUB2    PROC      FTABL ( )
        PFRMT     FTABL
```

As actual parameters may be passed:

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

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

1.4.4.2 *Assembler Subroutines*

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

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

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

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

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

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

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

The return values from these routines are:

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

A5 — Contains the data or literal end address.

A9 — Contains the data item or literal start address.

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

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

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

A8 — Contains the event control block address.

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

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

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

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

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

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

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

1.4.5 *Attach/Detach a device/file*

When a task wants to have exclusive access to a device or file, and locking out all other tasks from I/O at this device/file, the task has to execute an attach device instruction (DSC1, control code X'0E'). A time out value in multiples of 100 msec, must be specified for each attach to allow the monitor for supervising all attach requests and prevent (dead) lock situations. (DSC1, control code X'0B'). Time out value may be set to zero, then control is given immediately to the task which issued the request, with an indication in the extended status code whether the device of file is attached or not.

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

To release an attached device/file, the same task which issued the attach instruction, must perform a detach instruction (DSC1, control code X'0F').

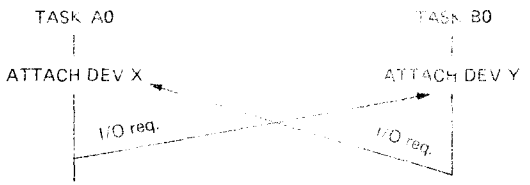
For an attached device all I/O requests from other tasks for this device are queued in a device queue and will be executed after the device is detached.

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

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

Such a situation will occur wh[illegible] time for each other to release attached resources.

Example: Device X is attached [illegible] to task B0. Task A0 issues [illegible] and B0 issues an I/O request for device X [illegible]



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

1.4.6 *Inter task communication*

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

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

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

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

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

When a task issues an inter task communication instruction, and no complementary instruction exists, the issued instruction is put into one of the four inter task communication queues, depending on whether the instruction was addressed to another task (RREAD, RWRITE) or unaddressed (READ, WRITE).
Two queues exist in the system, one for READ and one for WRITE (unaddressed). Only one of these queues may have one or more entries at any one time, since, as soon as they both contain an entry, the instructions are matched, communication takes place, and both instructions are completed.

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

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

If a READ, WRITE, RREAD or RWRITE instruction is to be supervised by the monitor in respect of time, a time out value should be set before the instruction is executed. Timing is set with the DSC1 instruction, with control code X'0B'. Different time out values in multiples of 100 msec, may be set for each instruction. These values are unique to the task which executed the time setting. The data-set-identifier, in the DSC1 instruction must refer to the corresponding DSET, for which the time must be set.
If no time out supervision is required, the binary data item in the DSC1 instruction, must be set to −1. If the value in this data item is set to zero, the request is completed immediately. No queueing is performed.

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

1.4.7 *Notation*

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

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

Examples:

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

1.4.8 *Instruction reference*

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

ABORT *Abort I/O request* **ABORT**

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

Type: I/O instruction

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

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

Condition mask:

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

Example: ABORT DSKBN

Intermediate code format:

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

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

ACTV *Activate* **ACTV**

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

Type: Scheduling instruction

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

Intermediate Code:

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

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

ADD *Add* **ADD**

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

Type: Arithmetic instruction

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

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

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

Condition mask:

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

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

Intermediate code format:

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

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

ASSIGN *Assign a file* **ASSIGN**

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

Type: I/O instruction

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

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

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

Condition register: =0 if assignment successful
=2 if assignment is unsuccessful

Condition mask:

0	1	2	3	4	5	6	7
SUCC	--	UNSUCC	--	SUCC	--	UNSUCC	Unconditional

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

Intermediate code format:

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

ASSIGN

Continued

ASSIGN

Intermediate code format: (continued)

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

Bytes 1 and 2 are filled by the system.

Byte 2 contains a reference to an external system routine.

operand-1 is a reference to a data set.

10/100 refers to the first data set.

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

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

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

Byte n contains a value filled by the translator.

operand 5, 6 are references to string data items.

ATTFMT *Attach Format* **ATTFMT**

Syntax: [statement-identifier] ␣ ATTFMT ␣ { format-list-identifier / data-item-identifier }

Type: Format control I/O

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

Condition register: Unchanged

Example: ␣ ATTFMT ␣ FRMT1

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.

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

B *Branch* **B**

Syntax: [statement-identifier] ␣ B ␣ [{equate-identifer, / condition mask,}] statement-identifier

Type: Branch instruction

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

Condition register: not changed.

Example:
B ␣ INP3
B ␣ 2,INP4

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

Byte 1	0	1	0	1	B	CND
Byte 2	displacement					

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

BBEOD *Branch on begin/end device* **BBEOD**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BBEOD ␣ DEVERR

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BE — *Branch on equal* — **BE**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BE ␣␣ EQUAL

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

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

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

Type: Branch instruction

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

Condition register: Unchanged.

Example: BEOF ENDOF1

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BERR *Branch on Error* **BERR**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BERR ␣ ERROR1

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BG *Branch on greater* **BG**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BG ␣ GREATER

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BL *Branch on less* **BL**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BL␣LESS

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BN *Branch on negative* **BN**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BN ␣ NEG

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNE — *Branch on not equal* — **BNE**

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

Type: Branch instruction

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

Condition register: Unchanged.

Example: BNE ␣ UNEQ

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNEOF *Branch on no End of file* **BNEOF**

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

Type: Branch instruction.

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

Condition register: Unchanged

Example: BNEOF ␣ NOTEOF

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNERR *Branch on no error* **BNERR**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BNERR ␣ NOERR

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNG *Branch on not greater* **BNG**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BNG␣NOTGRT

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNL *Branch on not less* **BNL**

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

Type: Branch instruction

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

Condition register: Unchanged.

Example: BNL␣NOTLESS

Intermediate code format: (long branch)

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

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

Intermediate code format. (short branch)

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

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

BNN *Branch on not negative* **BNN**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BNN ␣ NOTNEG

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNOK *Branch on not OK* **BNOK**

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

Type: Branch instruction

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

Condition register: Unchanged.

Example: BNOK ␣ NOTOKE

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNP *Branch on not positive* **BNP**

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

Type: Branch instruction

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

Condition register: Unchanged.

Example: BNP ␣ NOTPOS

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BNZ *Branch on not zero* **BNZ**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BNZ ␣ NONZER

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BOFL — *Branch on overflow* — **BOFL**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BOFL ␣ OVERFL

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BOK — *Branch on OK* — **BOK**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BOK ␣ OKE

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BP — *Branch on positive* — **BP**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BP ␣ POS

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

BZ — *Branch on zero* — **BZ**

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

Type: Branch instruction.

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

Condition register: Unchanged.

Example: BZ␣ZERO

Intermediate code format: (long branch)

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

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

Intermediate code format: (short branch)

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

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

CALL — *Call* — **CALL**

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

Type: Subroutine control instruction

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

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

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

Intermediate code format:

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

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

CBE *Compare and branch on equal* **CBE**

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

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').
If the contents of both operands are equal, the next instruction to be executed is found at the address specified by statement-identifie
If the contents of both operands are not equal, the instruction following the compare and branch equal (CBE) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch on equal.
Operand-1 and operand-2 must refer to the same type of data item decimal, binary or string.

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

Condition register: If both identifiers are references to numeric data items.
= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRING!
= 0 if (Operand-1) = (Operand-2)

Intermediate code format:

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

CBE *Continued* **CBE**

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

CBG *Compare and branch on greater* **CBG**

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

Type: Branch instruction.

Function: (Operand-1) ·· (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows.
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').
If the contents of operand-1 is not greater than the contents of operand-2, the instruction following the compare and branch on greater (CBG) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch on greater.
Operand-1 and operand-2 must refer to the same type of data item decimal, binary or string.

Example:
```
CBG    INLEN, $MIN, RDERR3
CBC    INLEN, CBINO, RDERR3
```

Condition register: If both identifiers are references to numeric data items.
= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRING!
= 0 if (Operand-1) = (Operand-2)

CBG *Continued* **CBG**

Intermediate code format:

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

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

CBL *Compare and branch on less* **CBL**

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

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

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

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

Condition register: If both identifiers are references to numeric data items.
= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRING!
= 0 if (Operand-1) = (Operand-2)

Intermediate code format:

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

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

CBNE *Compare and branch on not equal* **CBNE**

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

Type: Branch instruction.

Function: (Operand-1) ↔ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').
If the contents of operand-1 is unequal to the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
In all other cases the instruction following the compare and branch on not equal (CBNE) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch on not equal (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch.
Operand-1 and operand-2 must refer to the same type of data item – decimal, binary or string.

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

Condition register: If both identifiers are references to numeric data items.
= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRING!
= 0 if (Operand-1) = (Operand-2)

CBNE *Continued* **CBNE**

Intermediate code format:

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

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

CBNG *Compare and branch not greater* **CBNG**

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

Type: Branch instruction.

Function: (Operand-1) ·· (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows:
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').

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

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

Condition register: If both identifiers are references to numeric data items.
= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRINGI
= 0 if (Operand-1) = (Operand-2)

CBNG *Continued* **CBNG**

Intermediate code format:

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

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

CBNL — *Compare and branch on not less* — CBNL

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

Type: Branch instruction.

Function: (Operand-1) ≥ (Operand-2)

Description: The contents of operand-1 are compared with the contents of operand-2. The condition register is set according to the result of this comparison. When the two data items have a different size, the comparison will be executed as follows.
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').
If the contents of operand-1 is not less than the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
If the contents of operand-1 is not lesses than the contents of operand-2, the next instruction to be executed is found at the address specified by statement-identifier.
In all other cases, the instruction following the compare and branch on less (CBNL) will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the compare and branch on not less (incl. 4 bytes of the compare and branch) or 255 bytes after the compare and branch.
Operand-1 and operand-2 must refer to the same type of data item - decimal, binary or string.

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

Condition register: If both identifiers are references to numeric data items.
= 0 if (Operand-1) = (Operand-2)
= 1 if (Operand-1) > (Operand-2)
= 2 if (Operand-1) < (Operand-2)

Condition register: If both operands are of the type STRING or STRING!
= 0 if (Operand-1) ≥ (Operand-2)

CBNL *Continued* **CBNL**

Intermediate code format:

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

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

CLEAR — *Clear* — CLEAR

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

Type: Logical instruction.

Function: 0 → data-item-identifier

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

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

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

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

Condition mask:

0	1	2	3	4	5	6	7
=0	–	–	–	≠0	–	–	–

Intermediate code format:

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

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

CMP *Compare* **CMP**

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

Type: Arithmetic instruction

Function: (Operand-1) − (Operand-2)

Description: Operand-1 is compared with Operand-2.
The condition register is set according to the result of this comparison.
When the two data items have a different size, the comparison will be executed as follows:
a) for string data items the shortest item will be extended (by the interpreter) with blank characters (X'20').
b) for decimal data items the shortest item will be extended (by the interpreter) with zero digits (X'0').
Both operands must be the same type of data item – decimal, binary or string.

Condition register: If both operands are numeric or string data items:
= 0 if (operand-1) = (operand 2)
= 1 if (operand-1) > (operand 2)
= 2 if (operand-1) < (operand-2)

Condition mask:

0	1	2	3	4	5	6	7
Op1=Op2	Op1>Op2	Op1<Op2	–	Op1≠Op2	Op1⩽Op2	Op1⩾Op2	unconditional

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

Intermediate code format:

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

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

COPY *Copy* **COPY**

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

Type: String instruction

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

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

Condition register: Not significant.

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

Intermediate code format:

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

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

DELAY *Delay* **DELAY**

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

Type: Scheduling instruction

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

Condition register: Unchanged.

Example: DELAY DELTIM

Intermediate code format:

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

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

DETFMT *Detach format* **DETFMT**

Syntax: [statement-itenditier] ␣ DETFMT ␣

Type: Format control I/O.

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

Condition Register: Unchanged

Intermediate Code Format:

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

Bytes 1 and 2 are filled by the system.

DISPLAY **DISPLAY**

Syntax: [statement identifier] DISPLAY [control value]

data-item-identifier-1 [data-item-identifier-2 | literal constant]

Type: Format control

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

control value — Significance

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

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

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

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

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

DISPLAY *Continued* DISPLAY

Condition register: = 0 if OK
= 2 if ERROR

Condition mask:

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

Example: DISPLAY ␣ 0, LINE 3, LINE 12

Intermediate code format:

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

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

DIV *Divide* **DIV**

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

Type: Arithmetic instruction

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

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

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

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

Intermediate code format:

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

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

DLETE *Delete* **DLETE**

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

Type: String instruction

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

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

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

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

Condition register: Not significant.

Example: DLETE DIELD,P1,L1

Intermediate code format:

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

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

DSC1 *Dataset control one* **DSC1**

Syntax: [statement-identifier:] ␣ DSC1 ␣ [,NW,] data-set-identifier, {control value / equate-identifier} ,data-item-identifier

Type: I/O instruction

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

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

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

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

Condition mask:

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

Example: DSC1 DSSOPO,OFF,ALLAMP

Intermediate code format:

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

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

DSC1 *Continued* DSC1

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

[1] DL = logon disk file

Device dependent control information

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

PTS 6241 and 6242.

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

PTS 6232 and 6234

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

PTS 6233

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

PTS 6331:

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

PTS6236, PTS6271, PTS6272

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

DSC1 *Continued* DSC1

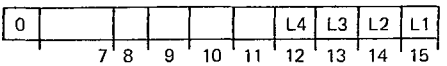
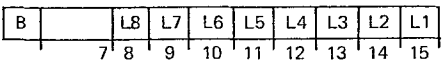
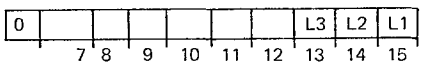
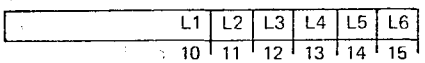
Control code	Device type	Data-item contains:	Significance
00	DC	0 7 8 15 0 0 | TASK Address 0 7 8 15 TC-Select address | TC-poll address	When issued from normal task When issued from a DC task
00	Badge card reader lamps 0 | L2 | L1 0 14 15 L2 L1 effect 0 0 no action 0 1 lamp is turned on (input from BCR) 1 0 not valid 1 1 flash lamps (input from PIN keyboard)		
01	L2 L1 effect 0 0 no action 0 1 if lamp on, turn lamp off; if lamp flashing, not valid; if lamp off, no action 1 0 if lamp on, no action; if lamp flashing, turn lamp on; if lamp off, no action 1 1 if lamp on, turn lamp off; if lamp flashing, turn lamp off; if lamp off, no action		
01	DC		Set status
02	DY	Number	A number of characters as specified in the binary data-item, are erased from the current cursor position. Only characters on the same line where the cursor is positioned can be erased. The cursor remains at the original position. The maximum number of characters that can be erased is as follows: PTS6344 – 80 PTS6351 – 36 PTS6365 – 40 PTS6386 – 40

DSC1 *Continued* **DSC1**

Device dependent control information

Control code	Device type	Data-item contains	Significance
06	DY	LINE NUMBER | COLUMN 0 7 8 15	Cursor is sent to line and column addressed in the binary data item X'7FFF' is the cursor home position. PTS 6241 (24 lines, 64 char per line optional 24 lines, 80 char per line) PTS 6242 (alpha-numeric version) — 8 lines, 80 char per line. PTS6385 1 line of 40 characters PTS6386 8 lines of 40 characters No information is erased
06	TV	number of line steps.	Position cursor/paperbook by giving the number of line feed steps (2 steps one line), in the binary data-item.
08	DL	Logical record number.	The status character is set to "FREE" on the record, the logical record number of which is in the binary data item. Delete is only allowed on a record which is under exclusive access. Exclusive access is released after function. (No check is performed to detect if the record was already "FREE")
0B	II, IO	Number	Set time in multiples of 100 msec for intertask communication, attach/detach device/file or data communication.
0B	SOP,DI KI	PTS 6241 and 6242 0 | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 7 8 9 10 11 12 13 14 15	Light positions corresponding with the one bits in the binary data item are lit once every second.

Device dependent control information:

Control code	Device type	Data-item contains:	Significance
		PTS 6232 and 6234  PTS 6233  If B=1, a buzzer is sounded at the keyboard.	
0B	SOP DI KI	PTS 6331:  PTS 6236:  	Lamp L1 is the left most lamp on each device.
0C	DL	Current Record Number (Data record)	Current Record Number of a data file is returned in the binary data item.
0D	DL	Current Record Number (Index record)	Current Record Number of a index file is returned in the binary data item
0D	TV,TJ (PTS6371)		

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

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

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

DSC1 *Continued* DSC1

Control code	Device type	Data-item contains:	Significance

The data items contains the parameter information, as follows:

0 3	4 7	8 11	12 15
L	NCV	CPJ	CPD

where : L is the case indicator;
if zero, no change is required;
if set to eight only upper case characters are required. Any code in the range /60–/7E is printed as the corresponding capital letter. If set to nine, both capitals and lower case are printed; the height of the capitals is reduced from 2.7 mm to 2.1 mm by using seven dots instead of nine.

NCV is set within the range 0–A for the national character variations, as shown in the tablet at the end of this DSC1 description.

CPJ and CPD are the character pitch for the journal and document stations respectively; if set to zero, no change is required.
The pitch may be 4, 5 or 6, corresponding with 15 char/inch, 12 char/inch, or 10 char/inch respectively.

If any of the parameters have an illegal value, none of the parameters will be set, and the request is completed with CR = 2 (Error).

If the printer is not operable for any reason, the request is completed with CR = 2 (Error); in this case the parameters are stored and sent to the printer when power is restored, but in practise they should be sent again, unless an XSTAT shows that this was the only cause of the CR being set to 2.

0E		Number	Attach a device or file, with a time out value in the binary data item. (Multiples of 100 msec). Time out zero is allowed; then control is immediately given back to the task which issued the request. Statuscode indicates whether the device or file is attached.

DSC1 *Continued* DSC1

Table of National Character Variations

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

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

DSC1 *Continued* DSC1

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

DSC2 — *Data set control two* — **DSC2**

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

Type: I/O instruction

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

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

Condition mask:

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

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

Intermediate code format:

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

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

DSC2 *Continued* DSC2

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

Control code X'11' Significance

This instruction is used to define the print layout and size of a document, by supplying a set of parameters describing the document. The number of sets is specified during system generation. The different document parameter sets are held in a table in the system, and can be referenced by an index having a starting value of zero for the first entry.
The first entry in the table is supplied with a set of standard parameters for A4 unfolded documents, which may be used if required. These are shown in a table at the end of this instruction description.
These parameters are included during system generation, and this instruction is only used to redefine a parameter set during application running, where a correction or change is necessary.
The number of characters to be transferred must be 14, 18 or 22, depending on the length of the parameter set to be replaced in the table (see below).
Data item identifier-1 refers to a binary data item, which must contain the index value for the required table entry.
Data item identifier-2 refers to a string data item which must contain the set of parameters to be replaced in the table.
All parameters must be supplied in ISO-7 code format.
If any of the parameters are missing or have an incorrect value, the request is completed with CR= 2 (Error), and the table in the system is not updated.

Parameter table entries :

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

DSC2 *Continued* DSC2

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

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

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

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

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

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

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

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

DSC2 *Continued* DSC2

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

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

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

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

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

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

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

DSC2 *Continued* DSC2

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

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

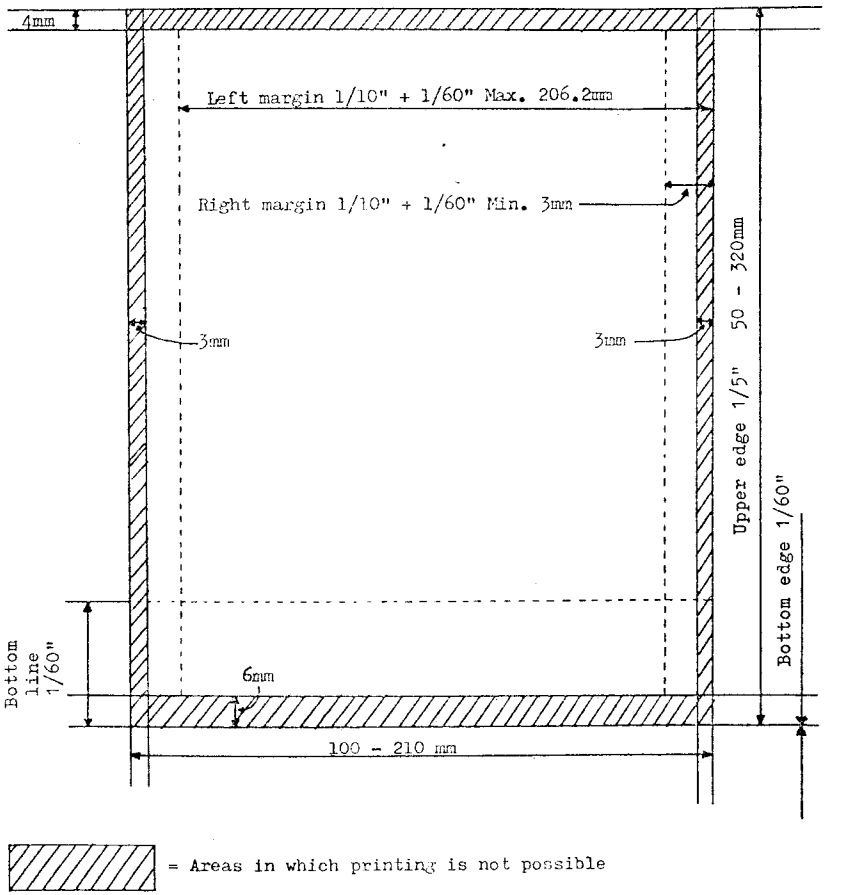
DW: Document width. This is only significant for document type 2, and is the width of the document in units of 1/10" (2.54mm).

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

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

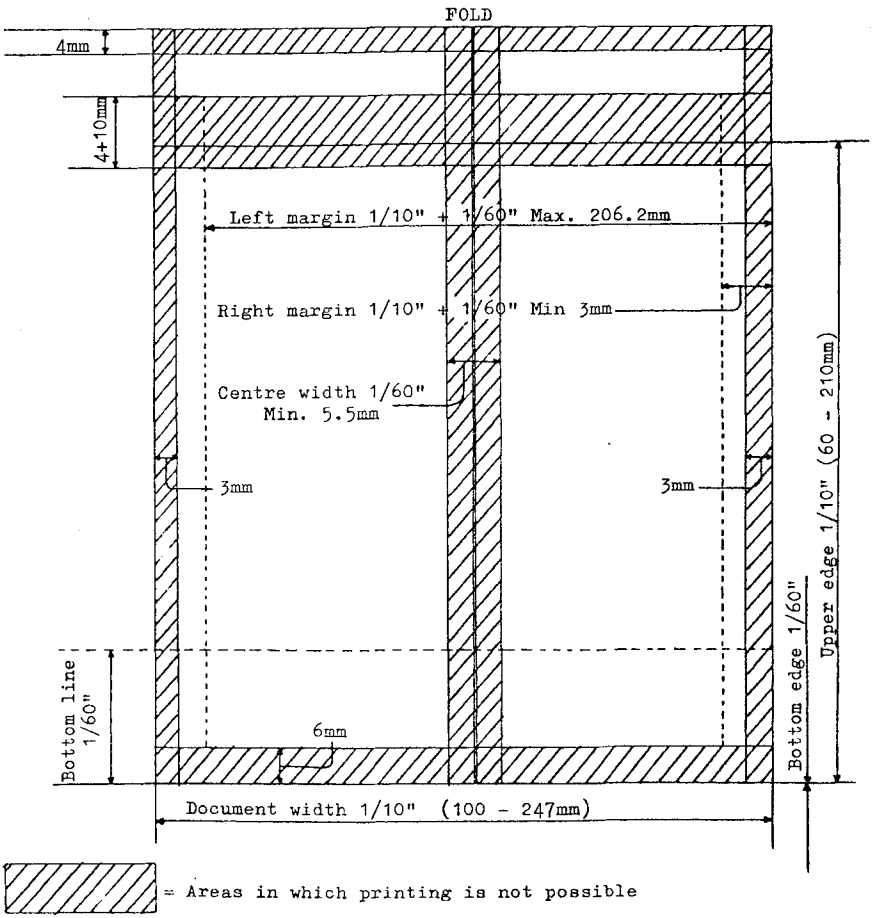
DSC2 *Continued* DSC2

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



DSC2 *Continued* DSC2

Diagram of parameters for document type 2 (Vertically folded)



DSC2 *Continued* DSC2

Table of National Character Variations

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

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

DSC2 *Continued* DSC2

Diagram of parameters for document type 3 (Horizontally folded)

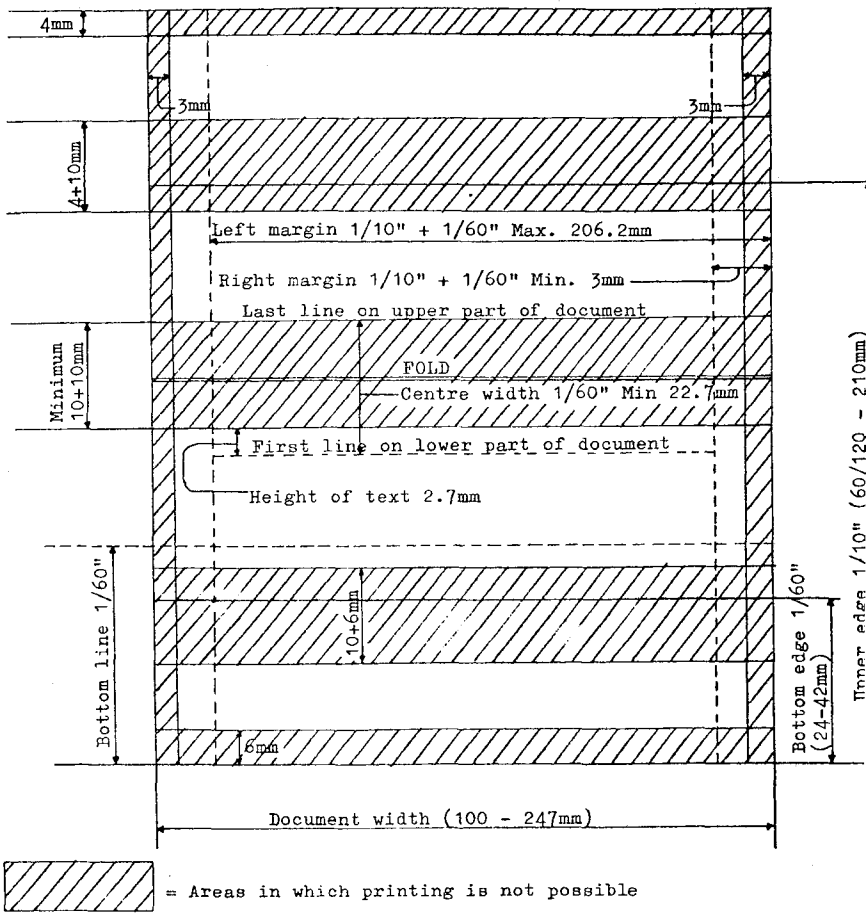


Table of Standard Document Parameters (entry zero in Parameter Table)

Parameter Type	Value	Description
DT	1	Unfolded document
TO	0	No timeout; printer will wait until document is inserted
LS	10	10/60" between each line on the document
NL	68	Number of lines is 68
BL	17	The distance from the bottom of the document to the bottom of the characters on the last line (number 68) is 17/60" = 7.2 mm.
MA	2	The margin is set 2/102 + 2/60" = 14/60" = 5.9 mm from the rightmost edge of the document.
MF	2	
LM	0	Print with right margin. The last character on each line is placed 5.9 mm from the rightmost edge of the document
CM	0	No critical margin; gives faster positioning
HP	0	Normal print pressure; this assumes multipart sets are not being used.
UE	58	58/5" = 11.6", the height of an A4 document
BE	0	No inner pages on the document (like passbooks)
DW		Not required
CW		

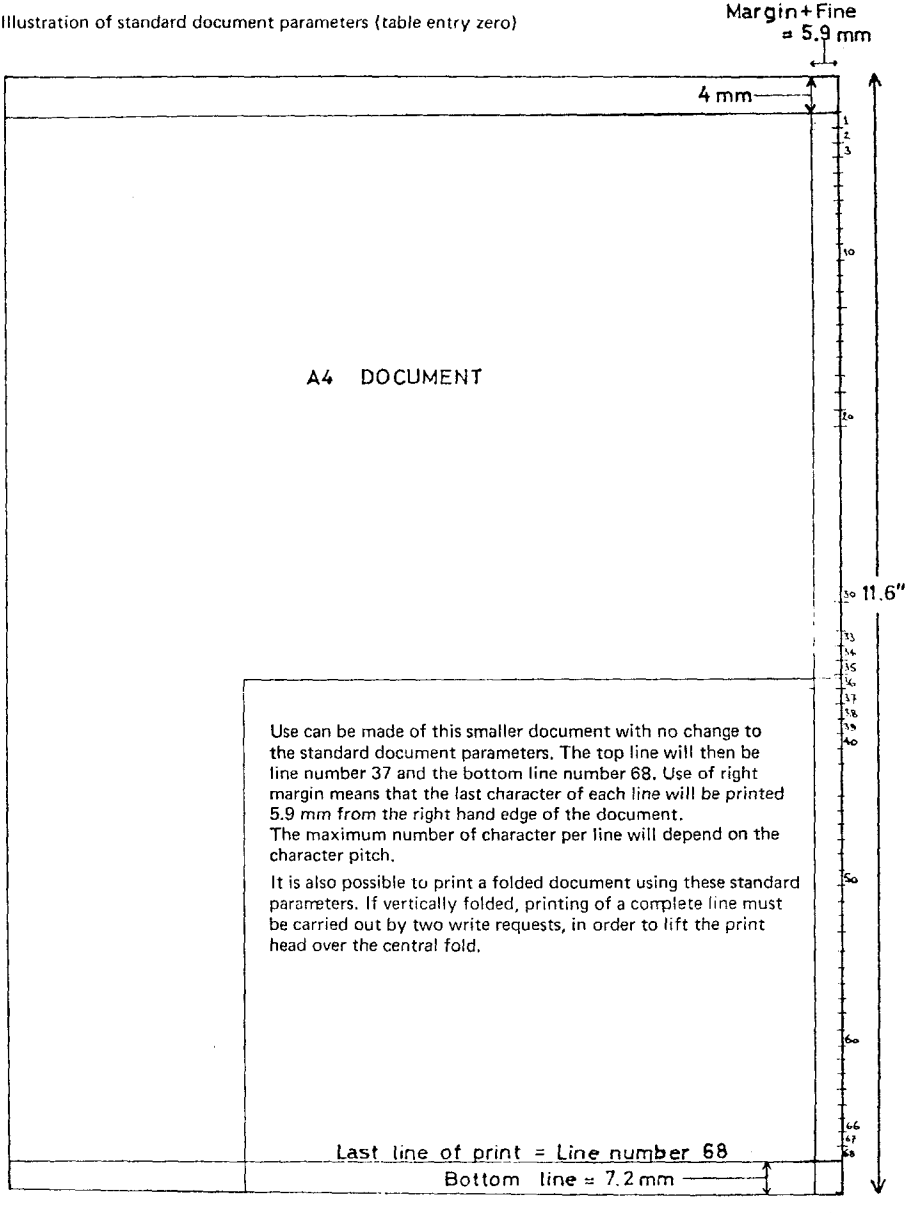
Note : this document uses right margin. This means that if a smaller document is used, printing may still take place, starting at a higher line number than 1 (i.e. lower on the page), and without using some of the lefthand print positions.
Thus a different document may be handled without the necessity for the user to send any document parameters; see illustration on next page.

DSC2

Continued

DSC2

Illustration of standard document parameters (table entry zero)



DUPL — *Duplicate* — DUPL

Syntax: [statement-identifier] ␣ DUPL ␣ data-item-identifier.

Type: Format control I/O

Description: The contents of the duplication data-item, as defined by the FKI-format list declaration, of the current input field is moved to the string data-item referenced by data-item-identifier.
A duplication data-item in a FKI-input field, may be of the type decimal or string.
The DUPL instruction uses the same conversion rules as the MOVE instruction for conversions from:

string ⟶ string
decimal ⟶ string.

Exception: When moving from string to string type of data item and the size of the receiving data-item is greater than the size of the sending data item, the remaining characters in the receiving data item will be X'00', instead of repeating the last character.

Condition register: = 0 Operation successfully performed
= 2 No duplication data-item associated with the current input field. (See FKI).

Condition mask:

0	1	2	3	4	5	6	7
SUCCESS	–	NO DUPL ITEM	–	$\overline{\text{SUCCESS}}$	DUPL ITEM	–	UNCON-DITIONAL

Example: DUPL ␣ DUPITEM

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Operand-1 is a reference to a string data item

DVR *Divide rounded* **DVR**

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

Type: Arithmetic instruction

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

Description: Operand-1 is divided by operand-2. The result is augmented by 0.5 and then rounded down. It is stored in operand-1. Operand-2 is unchanged. Both operands must be decimal or binary. Division by zero results in overflow and operand-1 is set to zero.

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

Intermediate code format:

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

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

DYKI *Display Keyboard input* **DYKI**

Syntax: [statement-identifier] ␣ DYKI ␣ data-item-identifier-1, key-table-identifier-1, key-table identifier 2, size-identifier, index-identifier, data-item identifier-2

Type: Format control I/O.

Description: Characters are read from the input data set as mentioned in the FMTCTL declaration in the data division, and stored in a string data item referenced by data-item-identifier 1 (Input buffer). The size of this buffer must be greater than the size as mentioned in the "MAXL" option, of the current input field, to allow the end-of-item key to be entered in the buffer too. Input is performed as a KIA-instruction with the input characters echoed on the output data set as mentioned in the FMTCTL declaration. Only the first input character is checked if it is present in the keytable referenced by key-table-identifier-1. The first four positions in the keytable have a predefined significance. (See below). If this character is not present in key table-1, the current input field on the display is filled with periods.
The second and following input characters are read and checked with key-table-2, from which the first four positions also have a predefined significance (See below). If an input character is present in the key-table, its position number (minus one), as declared in the key table, is returned in a binary data item referenced by index identifier. After completion of the transfer a converted key table index value is returned in this data-item which contents may be zero, a negative value or a positive value with the following meaning.

zero: Power failure has been present.

negative: A key lock switch has been turned.

positive: An index value ranging from 1 to (n-1) corresponding with the position number minus 1 in keytable-1 or keytable-2 is returned; if the transfer was correctly completed.

Note: Index value one corresponds with the second key code in the key table.
'n' is the number of key codes in the key table.
When no-end-of-item key is used to complete the transfer, the index value will be set to an undefined value outside the range -255 to +255.

If an illegal key code is received or the number as specified in MAXL is exceeded, a bell signal is sent to the display and input is restarted. After completion of the transfer a binary data item referenced by size identifier contains the number of characters transferred excluding the end of item key.

DYKI *Continued* DYKI

When an error occurs before the transfer is completed, an error code is returned in the binary data item referenced by data-item-identifier-2. This error code may be:

0 – no error

1 – number of characters received is less than the number specified in MINL.

2 – not used

3 – I/O error

4 – request aborted.

Expected predefined key table items in keytable-1 and keytable-2:

Position number in keytable	Significance
1	BACKSPACE. When this key code is entered the cursor is moved one position to the left, and a period is displayed in the new cursor position. If the first character position of the current input field is reached, the same function as CLEAR2 will be executed.
2	CLEAR1. The current input field is erased on the screen and its current input data item is cleared. Cursor is positioned at the first position of the next input field.
3	CLEAR2. The current input field is erased on the screen, and the cursor is positioned at the first position of the next input field.
4	EOI. General end of item key. Checks according to the number as mentioned in MINL is performed

Transfer ended when:

a) Any of the keys listed in keytable-1 (first position in the input field) or keytable-2 (second and following positions) except BACKSPACE, is received.

b) The maximum number of characters as defined by MAXL is reached and the current input field has the "NEOI" – flag set.

c) Power failure occurs

d) Keylock is turned

e) I/O error occurs.

DYKI *Continued* DYKI

Condition register: = 0 if OK (Error code in data item referenced by data item identifier-2 is zero)

= 2 if Error (Error code in data item referenced by data item identifier-2 is not zero)

Condition mask:

0	1	2	3	4	5	6	7
OK	--	ERROR	--	OK	--	ERROR	UNCON-DITIONAL

Example: DYKI␣BUFFER, KTB1, KTB2, SIZE, INDEX, ERRCODE

Intermediate code format:

Byte 1	0 0 1 1	0 0 0 0
Byte 2	external reference	
operand-1	data-item-identifier-1	
operand-2	key table identifier-1	
operand-3	key table identifier-2	
operand-4	size-identifier	
operand-5	index-identifier	
operand-6	data-item-identifier-2	

Bytes 1 and 2 are filled by the system.
Operand-1 is a reference to a string data item.
Operands-2, 3 are references to key tables.
Operands-4, 5, 6 are references to binary data items.

EDFLD *Edit Input Field* **EDFLD**

Syntax: [statement-identifier] ␣ EDFLD ␣ data-item-identifier-1, key table. identifier, size-identifier, index-identifier, data-item-identifier-2.

Type: Format control I/O

Description: Editing is performed in the string data item referenced by data-item-identifier-1 (buffer).
The size of this data item must be greater than the number mentioned in "MAXL" of the current input field. Depending on the contents of the binary data item referenced by size-identifier, the following operations are performed.

(size-identifier) = 0 The contents of the data item of the current input-field is moved to the data item referenced by data-item identifier-1. If the "REWRT" flag is set for the current input field, the contents of data-item-identifier-1 is displayed on the screen in its proper position without editing according to a picture definition.
The cursor is placed at the first character position of the field.

(size-identifier) ≠ 0 The cursor is placed at a character position, the number of which is contained in the binary data item referenced by size-identifier. "1" corresponds with the first character position.

Then characters are read from the input device, declared in the FMTCTL declaration, into the character positions in the buffer (data-item-identifier-1). The character positions correspond with the cursor position within the current input field. If a keycode is received which is present in the first four positions of key table, referenced by key-table-identifier, the corresponding function is executed and reading is resumed.

On a illegal keycode, a bell signal is sent to the output device (FMTCTL) and reading is resumed.

Expected predefined keytable items in keytable :

Position number in keytable	Significance
1	—► Non destructive space. Cursor is moved one position to the right. No action if cursor is at the right-most position of the current input field, or beyond the last significant character (i.e. at X'00' in the buffer).
2.	◄— Non destructive backspace. Cursor is moved one position to the left. No action if cursor is at the left-most position of the current input field.
3.	INS. Insert character. The characters from the current cursor position up to the last position in the field are shifted one step to the right. Any character shifted beyond the end of the line is dropped.

EDFLD *Continued* EDFLD

Position number in keytable	Significance
4	DEL. Delete character. The character at the current cursor position is deleted. Characters to the right of the current cursor position are shifted one step to the left. Cursor is not moved.
5	CLEAR1. Clear input field and input data item. Cursor is moved to the first position of the current input field. Terminate EDFLD. The contents of data item referenced by size identifier, is set to zero.
6	CLEAR2. The current input field is erased on the screen and the cursor is positioned, at the first position of the input field. Terminate EDFLD.
7	CLEAR3. Clear remaining positions in the field. The characters from the current cursor position, up to the last character in the field (inclusive) are cleared. Terminate EDFLD.
8	EOI. Common end of item. Contents of the binary data item referenced by index identifier, is set to three.
9 or higher	Terminate EDFLD.

Editing is terminated when:

a) A keycode is received, which is present in position 5 or higher, in the keytable.
b) Power failure has occurred.
c) A keylock switch is turned.
d) I/O error occurs.

After completion of this instruction, a value with following significance is returned in the binary data item referenced by index identifier:

zero: power failure has occurred.

negative: a keylock switch has been turned.

positive: an index in the range from

1 to (n-4) is returned, corresponding to positions 5 to N in the keytable (n is the keycode position in the keytable). Index value 3 is returned when the common EOI code, from position 8 in the keytable, is received.
In the binary data item referenced by size identifier is returned the effective length of the operation. The effective length is the number of resulting non-null characters in the buffer (data-item-identifier-1).
A null character has code X'00'.

In the binary data-item referenced by data-item identifier-2, is returned a code with following significance :

EDFLD *Continued* EDFLD

Contents	Significance
0	OK
1	The effective length is less than "MINL" (not set when CLEAR1)
2	not used.
3	I/O error
4	request aborted.

Condition register: = 0 if OK
= 2 if ERROR (The data item referenced by data-item-identifier-2, contains the error code).

Condition mask:

0	1	2	3	4	5	6	7
OK	–	ERROR	–	$\overline{OK}$	–	$\overline{ERROR}$	UNCON-DITIONAL

Example: EDFLD ␣ SPINPUT, SPKTAB3, SPBINW1, SPBINW2, SPBINW4.

Intermediate object code:

Byte 1	0 0 1 1	0 0 0 0
Byte 2	external reference	
operand-1	data-item-identifier-1	
operand-2	key table identifier	
operand-3	size-identifier	
operand-4	index-identifier	
operand-5	data-item-identifier-2	

Bytes 1 and 2 are filled by the system
Operand-1 is a reference to string data item.
Operand-2 is a reference to a key table.
Operands-3, 4, 5 are references to binary data items.

EDIT *Edit* **EDIT**

Syntax: [statement-identifier]␣EDIT␣ data-item-identifier-1,{data-item-identifier-2 / format-list-identifier}

Type: String instruction

Description: This instruction uses the format list to convert decimal and string data items into an edited string. The data items specified in the format list are edited according to the specified format and stored in a string data item indicated by operand-1.

Format-list-identifier is a reference to an edit format list which is composed of format declarations (FRMT, FCOPY, FMEL etc.) Instead of a format-list-identifier, operand-2 may be a reference to a string data-item. This data item must contain format-list characters as present in the format-literalpool. (output CREDIT linker). Item size must be great enough to contain these characters. The CALL FMOVE instruction may be used to fill the data-item.

Condition register: Not significant.

Example: EDIT FIELD, FORM1

Intermediate code format:

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

Byte 1 is the operation code (X'60' or X'61')
Operand-1 is a reference to a string data item.
L=1 operand-2 is a reference to a format list.
L=0 operand-2 is a reference to a string-data-item.

EDSUB *Edit Substring* **EDSUB**

Syntax: [tatement-identifier] ␣EDSUB␣ data-item-identifier-1, pointer-identifier, {data-item-identifier-2 / format-list-identifier}

Type: String instruction.

Description: Editing as specified in the formatlist is performed into a subfield of the string-data-item indicated by operand-1, beginning at pointer-identifier.
Upon completion, the binary-data-item indicated by pointer identifier is updated and points to a position immediately after the last position affected by the editing.
The first character in the string data item is counted as zero when setting the pointer.
Instead of a format-list-identifier, operand-3 may be a reference to a string data-item. This data-item must contain format-list characters as present in the format-literalpool. (output CREDIT linker). Item size must be great enough to contain these characters. The CALL FMOVE instruction may be used to fill the data-item.

Condition register: Unchanged.

Example : EDSUB ␣ BUF, P1, FRM001

Intermediate code format:

Byte 1	0 1 1 0	1 1 0	L
operand-1	data-item-identifier-1		
operand-2	pointer-identifier		
operand-3	format-list-identifier		

Byte 1 is the operation code (X'6C', X'6D')
operand-1 is a reference to a string-data-item.
operand-2 is a reference to a binary-data-item.
L=1 operand-3 is a reference to a format list.
L=0 operand-3 is a reference to a string data item.

EDWRT *Edit and Write* **EDWRT**

Syntax: [statement-identifier,] EDWRT data-set-identifier,
{format-list-identifier / data-item-identifier}

Type: I/O instruction.

Description: The data items specified in the format list are edited into a buffer with a format as indicated by the format list. After editing, the buffer is output to the device indicated by the data set.
Format-list-identifier is a reference to an edit format list which is composed of format declarations (FFMT, FDOPY, FMEL, etc.).
Instead of a format-list-identifier, operand 2 may be a reference to a string data-item. This data-item must contain format list characters as present in the format list (output of the CREDIT linker). Item size must be great enough to contain these characters. The CALL FMOVE instruction may be used to fill the data-item.
Data-set-identifier is a reference to the relevant data set. .NW, indicates that the no wait option is required.
One buffer per output data set is allocated to the program. The size of the buffer, is defined in the data set declaration. An EDWRT operation on a data set should be completed before another EDWRT is executed on that data set.
The first two bytes in the buffer can contain a control character. The first byte must always be unequal to zero and the second byte contains the control character. The function of the control character is device dependent. The control character may have the following values:

- General Terminal Printer or line printer:
  X'2B' Print the line without advancing the paper.
  X'30' Advance two lines before printing.
  X'31' Skip to top of form before printing. (only for line printer)
  Other codes: One line feed and carriage return is executed before printing.

Special characters allowed in the user buffer and not restricted to the first word in the buffer:
X'11' Tabulation character. This character should be followed by two ISO-7 digit characters giving the tabulation position. (only for GTP)

- Teller terminal printer (PTS6222, PTS6223):
  Voucher/passbook printing.
  X'2B' Print the line without advancing the paper.
  X'30' Advance two line steps before printing.
  X'31'–X'39' Advance paper 1–9 line steps before printing.
  Other codes: One line step and carriage return is executed before printing.

EDWRT *Continued* EDWRT

Journal/tally roll printing.
X'30' Advance two line steps before printing. (two steps = one line feed)
Other codes: One line step is executed before printing.

Special characters allowed in the user buffer:
X'09' The printhead is moved to the rightmost print position of the voucher. This character should be present in the last buffer position.
X'0D' The printhead is moved to the rightmost position of the journal station. This character should be present in the last buffer position.

Video display or plasma display
X'2B' The text is displayed from current cursor position.
X'30' Cursor is advanced two lines and positioned at the beginning of the line, before the text is displayed.
X'31' Erase display and position cursor on home position before the text is displayed.
Other codes: Advance cursor one line before the text is displayed.

- Teller terminal printer PTS6371
  The control character present in the second character of the buffer, as follows:
  /2B – printing is carried out from the last position of the previously printed line on this device. However, if the character pitch has been set, or if positioning has been carried out to the same line, since the previous line was printed, the printing will be from the tabulation position on the present line.
  /30 – the paper is advanced two lines, and the printing carried out from the tabulation position.
  /31 – journal: the paper is advanced three lines and the printing carried out from the tabulation position. This will make the previously written data readable through the window on the journal station.
  – document: printing is started from the tabulation position on line 1.
  Any other value in the control code will cause one line feed before printing from the tabulation position.
  The requested length must include the two bytes used for the control code, but if it is two, only the action specified by the control code is carried out.
  The maximum line length on the two print stations is limited to the following, based on normal character width.

	Journal	Document
10 characters/inch	33	80
12 characters/inch	40	
15 characters/inch	50	

One expanded character equals two normal characters.

EDWRT *Continued* EDWRT

Special characters allowed in the user buffer and not restricted to the first word in the buffer :

Characters Valid for All Displays

/AE : Displayed as point (/E2)
/11 : Tabulation character. This character should be followed by two ISO-7 digits giving the tabulation position.
/07 : Bell is sent to the display.

Characters Valid for PTS 6344 only

/12 : Underline start. Output of characters which follow this character are provided with underline.
/13 : Underline stop. Output of characters which follow after this character are not provided with underline. Underline stop mode will also appear at request end.
/14 : Fast output. First character following /14 will be transmitted in fast output mode up to requested length. Note that cursor will remain unchanged.
/1C : Data to keyboard.
/1D : Master clear to keyboard
/1E : Low intensity start. Output of characters which follow after this character, are displayed at low intensity.
/1F : Low intensity stop. Output of characters which follow after this character are displayed at normal intensity. Normal intensity mode will also appear at request end.

Characters Valid for PTS 6371 printer only

/12 : Underline start. Output of characters which follow this character are provided with underline.
/13 : Underline stop, Output of underlined characters stops. Underlining also stops at request end .
/19 : Start / Stop expanded character mode. Characters following the first occurrence of this character in the buffer are printed as double width characters, until the next occurrence of this code in the buffer.
/1A : Each character in the range /30 – /3C which is preceded by this code is printed as an OCR–A character. Any other legal character preceded by this code is printed as a space.
/1B : Each of the characters described below, which is preceded by this code, is printer as a special character. Any other legal code that is preceded by this code is printed as a space.
Codes /20–/29 are for use when the National Character Variation currently in use (see DSC1) does not contain the character required. They are printed as Space, §, @, #, ◇, £, Space, ::, → and ↓ respectively.
Codes /30–/39 are printed as numerics with a greater width than normal, and are more the size of alphabetic characters.
Codes /3A–3F are logotypes, defined by the user. The character generator for these codes is a separate unit which must be in the printer. If it is not, these characters are printed as spaces.

EDWRT *Continued* EDWRT

/AE : Each character in the range /30—/39 which is preceded by this code is printed as a roomless point numeric.
Any other legal character preceded by this code is printed as a space.

Condition register: = 0 if I/O successful (OK)
= 1 if End of file (EOF)
= 2 if Error (ERR)
= 3 if Begin or End of Device (BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	OK	EOF	ERR	Uncondi-tional

Example: EDWRT DSTPTR, FRM001

Intermediate code format:

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

Bytes 1 and 2 are filled by the system. Byte 2 contains a reference to an external system routine.
W is the wait bit.
W = 0 no wait
W = 1 wait
Operand-1 is a reference to the relevant data set.
10/100 refers to the first data set.
L = 0 operand-2 is a reference to a string-data-item.
L = 1 operand-2 is a reference to a format list.

ERASE — *Erase* — ERASE

Syntax: [statement-identifier] ␣ ERASE ␣ control value,

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

Type: Format control I/O

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

Control Value	Significance
0	The lines, ranging from the line number contained in the binary data item, referenced by data-item-identifier-1, to the line number contained in the binary data item referenced by data-item-identifier-2 are erased on the screen. When the second line number (referenced by data-item-identifier-2) is zero, then all lines of the current format list are erased starting at the line number contained in the data-item referenced by data-item-identifier-1. Both data items may contain the same line number.
1	All input fields (FKI+FINP) of the current format list with an input field number ranging from the number contained in the binary data item referenced by data-item-identifier-1 up to the number contained in the binary data-item referenced by data-item-identifier-2 are erased on the screen.
2	As control value 1, but also data-items belonging to the input field are cleared.
3	As control value 1, but only data-items belonging to the input field are cleared.
4	As control value 1, but erasing is not performed on input fields with the "NCLR" flag set.
5	As control value 2, but erasing is not performed on input fields (and the corresponding data items) with the "NCLR" flag set.
6	As control value 3, but no resetting on to the input fields belonging data items is performed, which have the "NCLR" flag set.
9	As control value 1
10	As control value 2
11	As control value 3
12	As control value 4
13	As control value 5
14	As control value 6

Note: These control values are similar to the control values 1 up to including 6, but only FKI-type input fields are taken into account.

The last field number to be erased may also be indicated by a literal constant of the type binary.

ERASE *Continued* **ERASE**

Condition register: = 0 if OK
= 2 if ERROR

Condition mask:

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

Example: ERASE, SPBINW1, = '0'

Intermediate object code:

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

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

EXIT *Exit* **EXIT**

Syntax: [statement-identifier] ␣EXIT

Type: Scheduling instruction.

Description: Execution of the task is terminated, but may be restarted by the activate instruction.

Condition register: Not significant.

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.

GETABX — *Get Current input field number* — GETABX

Syntax: [statement-identifier] ␣ GETABX ␣ data-item-identifier

Type: Format control I/O

Description: The number of the current input field is returned in a binary data item referenced by data-item-identifier. When no input field is current, the content of the binary data item will be zero. The field type is indicated in the condition register.

Condition register:
= 0 The current input field is an FKI-type field
= 1 The current input field is an FINP-type field
= 2 No input field is current.

Condition mask:

- 0	1	2	3	4	5	6	7
FKI-type	FINP-type	NO CURRENT INP FLD	—	no FKI type	no FINP type	current inp fld.	UNCON-DITIONAL

Example:

```
GETABX FLDNUM.
```

Intermediate object code:

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

Bytes 1 and 2 are filled by the system.
operand-1 is a reference to a binary item.

GETCTL *Get control value* **GETCTL**

Syntax: [statement-identifier] ␣ GETCTL ␣ control value,

data-item-identifier

Type: Format control I/O.

Description: One of the values, following the options APPL, MAXL, MINL or SCHK, from the current input field is transferred to a binary data item, referenced by data-item-identifier. Options are specified in the FKI– or FINP– format list declarations. Value zero returned if the requested option is not defined in the FKI or FINP description.

control value	significance
0	The "APPL" value is transferred.
1	The "MAXL" value is transferred.
2	The "MINL" value is transferred.
3	The "SCHK" value is transferred.

Condition register: Unchanged.

Example: GETCTL ␣ 3, CHECK

Intermediate object code:

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

Bytes 1 and 2 are filled by the system
Operand-1 is the control value
Operand-2 is a reference to a binary data item.

GETFLD *Get input field* **GETFLD**

Syntax: [statement-identifier] ␣ GETFLD ␣ control value,

data-item-identifier-1, data-item-identifier-2

Type: *Format control I/O*

Description: The input field, of the current format list, which input field sequence number is contained in the binary data-item referenced by data-item-identifier-1, becomes current. The field sequence numbering to be used is specified in control value, which must be a decimal value
0 for FKI-input field sequence numbering.
1 for FINP-input field sequence numbering,
2 for all input field sequence numbering. When the data-item-identifier-1 refers to a binary data-item with contents zero, the last input field of the specified type (in control value) will become current.
If, before the execution of this instruction any empty compulsory field was found (its corresponding data-item is empty and in the FKI-field the muster enter flag ME, was set), then after execution of this instruction the number of this compulsory field will be returned in a binary data-item referenced by data-item-identifier-2 and the condition register is set.
On a successful operation this data item contains zero.

Condition register:
= 0 Operation successfully performed, no empty, compulsory field was found. (Compulsory field is defined in the FKI input field declaration).
= 2 The addressed input field sequence number was not found within the current format list.
= 3 Operation successfully performed but an empty compulsory field was found.

Condition mask:

0	1	2	3	4	5	6	7
OK	–	ERR	EMPTY	$\overline{\text{OK}}$	–	$\overline{\text{ERR}}$	Uncon-ditional

Example: GETFLD 0, INPF1, ERFLD

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Operand-1 is the control value 0, 1 or 2
Operands-2, 3 are references to binary data items.

GETID *Get task identifier* **GETID**

Syntax: [statement-identifier] ␣ GETID ␣ data-item-identifier

Type: Scheduling instruction.

Description: The current task identity is transferred to a data-item indicated by data-item-identifier.
The data-item may be of the type binary or string. In case of a string data-item only the first two character positions are affected.

Condition register: Unchanged.

Example: GETID, TASKID

Intermediate code format:

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

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

GETTIME *Get clock* **GETTIME**

Syntax: [statement-identifier] ␣ GETTIME ␣ data-item-identifier

Type: clock control.

Description: The current time of the system clock is returned in a string data item indicated by data-item-identifier.
The string data item must have a length of at least six characters.
The time is returned as ⌊H⌋⌊H⌋⌊M⌋⌊M⌋⌊S⌋⌊S⌋

H = hour
M = minute
S = second

Condition register: Unchanged.

Example: GETTIME TIME

Intermediate code format:

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

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

IASSIGN *Assign index file* **IASSIGN**

Syntax: [statement-identifier] ␣ IASSIGN ␣ data-set-identifier, data-item-identifier, file-name-identifier-1, file-name-identifier-2, volume-name-identifier

Type: I/O instruction

Description : The index file name (8 bytes including trailing blanks), present in the string-data-item referenced by file-identifier-1, is assigned to the data file referenced by data-set-identifier. The file code in the data set, which is already used for the data file assignment, now determines to which data file this index file will be assigned. The master index file, which name (8 bytes) is contained in the string-data-item referenced by file-name-identifier-2, is read into memory. Volume name identifier refers to a string-data-item (6 characters inclusive trailing blanks) in which is a reference to the volume on which master index file and index file are present. Maximum four index files may be assigned to one data file using different file codes. Index files must be assigned as common files.

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

0 assignment successful performed
-1 request error
1 Disk I/O error
2 No free entry in common device table.
3 Not sufficient memory space available for master index or file descriptor blocks
4 Volume name unknown
5 File already assigned from this task
6 File name unknown
7 File section missing or found twice
8 Faulty disk format
9 More than 4 extents exist
10 No data file assigned.
11 4 index files already assigned
12 Size of disk buffers not sufficient
13 Request busy. Reissue request.

Condition register: = 0 if assignment successful
= 2 if assignment unsuccessful

Condition mask :

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

Example: IASSIGN ␣ DFILE, ERRCODE, INDXFIL, MIXFIL, VOLNAM

IASSIGN *Continued* IASSIGN

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	0	0	data-set-identifier					
operand-2	data-item-identifier							
operand-3	file-name-identifier-1							
operand-4	file-name-identifier-2							
operand-5	volume name-identifier							

Bytes 1 and 2 are filled by the system.
Byte 2 contains a reference to an external system routine.
Operand-1 is a reference to a data set.
10/100 refers to the first data set.
Operand-2 is a reference to a binary data item
Operands-3, 4, 5 are references to string data items.

IB *Indexed branch* **IB**

Syntax: [statement-identifier] ␣ IB ␣ index-identifier { , statement-identifier / , external-identifier } ...

Type: Branch instruction.

Description: A branch is made to one of the identifiers in the identifier list according to the contents of the data item specified by index-identifier. The first identifier in the list corresponds with the index value one. If the index is zero, or greater than the number of identifiers in the list, the instruction following the indexed branch is executed.

Condition register: Not significant.

Example: IB INDEX, SYS20, SYS40

Intermediate code format: (long branch)

Byte 1	0	0	1	1	0	0	1	0
operand-1	index-identifier							
Byte n	list length							
operand-2	statement-identifier-1							
operand-3	statement-identifier-n							

Byte 1 is the operation code (X'32').
Operand-1 refers to a binary data item.
Byte n is filled by the CREDIT translator and contains the number of identifiers present in the address list.
Operands-2,3 etc. contain an index to a branch address table (T:BAT).

Intermediate code format: (short branch)

Byte 1	0	0	1	1	0	1	1	B
operand-1	index-identifier							
Byte n	list length							
Byte n+1	displacement-1							
Byte n+2	displacement-n							

Byte 1 is the operation code (X'36', X'37').
B = 0 forward branching
B = 1 backward branching
Operand-1 refers to a binary data item.
Byte n is filled by the CREDIT translator and contains the number of identifiers present in the address lits.
Bytes n+1, n+2 contain a displacement.

IINS *Indexed Insert* **IINS**

Syntax: [statement-identifier] ␣ IINS ␣ [,NW,]
data-set-identifier, data-item-identifier-1,
data-item-identifier-2

Type: I/O - instruction.

Description: The data record, present in the string data-item referenced by data-item-identifier-1, is written as a new record to the data file referenced by data-set-identifier and all associated index files are updated. All index files must be assigned (as common files) when this instruction is executed. The new data record will be written after the last record, pointed by last record number pointer (LRN), in the file. The last record number pointer is updated. If an index record with the same key already exists, the new record is placed before the old one.
In the binary-data-item referenced by data-item-identifier-2 is returned the number of remaining records in the data file. If this number is greater than 32.767, 32.767 is returned.
When in the status code bit 10, "End of medium" is obtained, one index record is lost, and the index files must be rebuilt
Bit 3 "End of File" can be used as a warning for this situation.

Condition register:
= 0 if I/O successful (OK)
= 1 if End of file (EOF)
= 2 if error (ERR)
= 3 if Begin or End of device (BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	Uncondi-tional

Example: IINS DSDF1, BUF1, FRNUM

Intermediate object code:

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

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine
W is the wait bit
W=0 no wait
W=1 wait
operand 1 is a reference to a data set.
10/10 refers to the first data set.
Operand 2 is a reference to a string data item.
Operand 3 is a reference to a binary data item.

INSRT *Insert* **INSRT**

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

Type: String instruction.

Function: (Operand-4) (pointer-identifier-2) $\xrightarrow{\text{inserted}}$ Operand-1 (pointer-identifier-1)

Description: Starting at pointer-identifier-2, the contents of operand-4 are inserted into operand-1 beginning at pointer-identifier-1.
The number of characters to be inserted is given in the data item specified by size-identifier. This insertion is accomplished by shifting the original contents of operand-1 to the right starting at pointer-identifier-1.

If a non-space or non-zero character is shifted out of operand-1, the condition register is set to overflow. Each character shifted out is lost.

The first characters of operand-1 and operand-4 are counted as zero when setting the pointers.

Operand-1 and operand-4 must be string data items.

Condition register: = 3 if Overflow

Condition mask:

0	1	2	3	4	5	6	7
–	–	–	over-flow	–	–	–	–

Example:
```
INSRT   TEXT1,P1,LNGTH,TEXT2,P2
```

Intermediate code format:

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

Byte 1 is the operation code (X'64').
Operands-1,4 are references to string data items.
Operands-2,3,5 are references to binary data items.

INV *Invert* **INV**

Syntax: [statement-identifier] ␣ INV ␣ data-item-identifier

Type: Logical instruction

Function: $(\overline{\text{data-item-identifier}}) \rightarrow$ data-item-identifier

Description: The content of data-item-identifier is inverted (replaced by complement).

Date-item-identifier must refer to a boolean data item (length 1 bit). The condition register is set according to the *previous* value of data-item-identifier.

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

Condition mask:

0	1	2	3	4	5	6	7
DI=0	–	–	–	DI≠0	–	–	–

Intermediate code format:

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

Byte 1 is the operation code (X'42').
Operand-1 is a reference to a boolean data item.

IREAD *Indexed Random Read* **IREAD**

Syntax: [statement-identifier] IREAD [.NW,] [.NEA,] data-set-identifier, data-item-identifier-1, size-identifier, data-item-identifier-2

Type: I/O instruction

Description: A data record is read from the data file indicated by data-set-identifier and stored in a string data-item indicated by data-item-identifier-1. The string-data-item indicated by data-item-identifier-2 must contain the symbolic key of the desired record. The number of requested bytes is put in the binary data-item-referenced by size-identifier, which on completion of this instruction will contain the number of bytes transferred.
.NW indicates that no wait option is required.
.NEA indicates that exclusive access should not be set for this record.
On a successful read the accessed record is available for this task under exclusive access.
Exclusive access is automatically released after:
- a write or rewrite of the record.
- a delete function.

Exclusive access may be released explicity by the "Release exclusive access" function.
The current record number (CRN) will point to the current data record and a CRN will point to the current index record.

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

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	OK	EOF	ERR	Unconditional

Example: IREAD .NEA, DSDK1, BUF1, SIZE, KEY

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	EA	data-set-identifier					
operand-2	data-item-identifier-1							
operand-3	size-identifier							
operand-4	data-item-identifier-2							

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine.
W is the wait bit.
EA is the exclusive access bit.
W=0 no wait EA=1 no exclusive access
W=1 wait EA=0 exclusive access
Operand-1 is the reference to the data set.
10/100 refers to the first data set.
Operands-2,4 are references to string data items.
Operand-3 is a reference to a binary data item.

IRNEXT *Indexed Read Next* **IRNEXT**

Syntax: [statement-identifier] ␣ IRNEXT ␣ [.NW,] [.NEA,] data-set-identifier, data-item-identifier-1, size-identifier

Type: I/O instruction

Description: The data-record, with the symbolic key following the previous symbolic key in the index file, is read when the instruction executed before was an indexed random read, indexed insert or indexed read next.
The contents of the data-record will be stored in the string data-item referenced by data-item-identifier-1. Data-set-identifier refers to a data-file. The number of requested bytes is put in the binary data-item referenced by size-identifier, which on completion will contain the number of bytes transferred.
.NW indicates that no wait option is required
.NEA indicates that exclusive access should not be set for this record.
Exclusive access is automatically released after:
- a write or rewrite of the record
- a delete function.

Exclusive access may be released explicity by the "Release exclusive access" function. The current record number (CRN) will point to the current data record and a CRN will point to the current index record.

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

Condition mask:

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

Example:
```
IRNEXT    DSDK1, BUF1, SIZE
```

IRNEXT *Continued* **IRNEXT**

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	EA	data-set-identifier					
operand-2	data-item-identifier-1							
operand-3	size-identifier							

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine
W is the wait bit.
EA is the exclusive access bit.
W=0 no wait EA=1 no exclusive access.
W=1 wait EA=0 exclusive access
Operand-1 is a reference to a data set
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a binary data item.

IRWRITE *Indexed Rewrite* **IRWRITE**

Syntax: [statement-identifier] ␣ IRWRITE ␣ [.NW,]
data-set-identifier, data-identifier-1,
data-item-identifier-2

Type: I/O instruction.

Description: The data-record indicated by its logical record number, which is present in a binary or decimal data-item referenced by data-item-identifier-2, will be overwritten with the contents of the buffer referenced by data-item-identifier-1, except for the key field. Data-set-identifier refers to the data file to be processed. The record must be under exclusive access, which is released after a successful rewriting of the record. Also all index files must be assigned (as common files) when this instruction is executed. When the key areas in the new data record and the old one, are not the same, bit 1 (key not found) will be set in the status code.
.NW indicates that no wait option is required.

Condition register:
= 0 if I/O successful (OK)
= 1 if End of File (EOF)
= 2 if Error (ERR)
= 3 if Begin or End of device (BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	Unconditional

Example: IRWRITE ␣ DSDK1, BUF1, RECNR

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine
W is the wait bit.
W=0 no wait
W=1 wait
Operand-1 is a reference to a data set.
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a binary or decimal data item.

KI *Keyboard input* **KI**

Syntax: [statement-identifier]␣KI ␣[.NW,][.NE,] data-set-identifier, data-item-identifier, key-table-identifier, size-dientifier, index-identifier

Type: I/O instruction.

Description: Alphanumeric characters are read from the keyboard indicated by data-set-identifier and stored in the string data item indicated by operand-4. The task waits, until transfer is completed.

The number of requested characters is given in the data item specified by size-identifier, which on completion of input will contain the number of characters transferred. The data item specified by index identifier is filled with the position number of the terminating character in the key table. The first character of the key table is counted as one. Key-table-identifier refers to the relevant key-table. .NW and .NE indicate that the no wait and no echo options are required.

KI can also be used to read the SOP switches. In this case the pointer contains the position number of the pressed switch. The rightmost switch on the SOP panel is counted as one.

Transfer of alphanumeric characters is ended if:

1) One of the terminating characters listed in the key table is input.
2) A character neither alphanumeric nor listed in the key table is input.
3) The size of the string data item is reached.
4) Power failure occurs.
5) Requested number of characters is reached.
6) A keylock switch is turned.

In case 2), 3) and 5) above the pointer will contain an undefined value and the condition register will be set to ERROR.
In case of a power failure the pointer is set to zero and no indication is given in the condition register.
All character positions not affected by the input are set to X'00'.
In case 6) the index value will be negative, thus indicating that a key-lock switch is turned.
The possible negative values in the index for keyboards PTS6236, 6271 and 6272 are:

−1: key-lock no.4 turned OFF
−2: key-lock no.3 turned OFF
−3: key-lock no.2 turned OFF
−4: key-lock no.1 turned OFF
−5: key-lock no.4 turned ON
−6: key-lock no.3 turned ON
−7: key-lock no.2 turned ON
−8: key-lock no.1 turned ON

If all keys are OFF, the keyboard is considered to be inactive.

KI *Continued* KI

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

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	unconditional

Example: KI DSKB,INBUF,KTAB1,INLEN,INDEX

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	E	data-set-identifier					
operand-2	operand-4							
operand-3	key-table-identifier							
operand-4	size-identifier							
operand-5	index-identifier							

Bytes 1 and 2 are filled by the system. Byte 2 contains a reference to an external system routine.
Byte 3 Bit 0 is the wait bit.
W is the wait bit
W=0 no wait
W=1 wait
E is the echo bit
E=0 no echo
E=1 echo
Operand-1 is the reference to the relevant data set.
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a key table which is assumed to be literal.
Operands-4,5 are references to binary data items.

LB *Long branch* **LB**

Syntax: [statement-identifier] ␣LBL␣ [{ equate-identifier, / condition-mask, }] { statement-identifier / external-identifier }

Type: Branch instruction.

Description: The next instruction to be executed is indicated by operand-2, when operand-1 matches the contents of the condition register. Otherwise the instruction following the long branch instruction will be executed.

If operand-1 is omitted, an unconditional branch (value 7) is generated.

Condition register: Not changed.

Example:
```
LB    SYSOPN
LB    3,SYSCLOS
```

Intermediate code format:

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

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

MATCH *Match* **MATCH**

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

Type: String instruction.

Function: (Operand-4) ·· (Operand-1)
(pointer-identifier-2) (pointer-identifier-1)

Description: This instruction searches the specified part of operand-1 in an attempt to find a match with the specified part of operand-4. The parts of operand-1 and operand-4 involved are defined by their respective pointer-identifier and size-identifier. The search commences at the character in operand-1 indicated by pointer-identifier-1, and continues for the number of characters specified in size-identifier-1. The characters in operand-4 to be searched for begin at the position specified by pointer-identifier-2. The number of characters to be searched for is specified by size-identifier-2.

If a match is found, pointer-identifier-1 will contain the address within operand-1 at which the match occurs and the condition register is set to *zero*. (Equal).

If no match occurs, pointer-identifier-1 will have an undefined value.

Operand-1 and operand-4 must be string data items.

The first characters in operand-1 and operand-4 are counted as zero.

Condition register: = 0 if match.

Condition mask:

0	1	2	3	4	5	6	7
EQUAL	–	–	–	UNEQUAL	–	–	–

Example:
```
MATCH     TEXT1,P1,L1,TEXT2,P2,L2
```

MATCH *Continued* **MATCH**

Intermediate code format:

Byte 1	0 1 1 0	1 0 0 0
operand-1	data-item-identifier-1	
operand-2	pointer-identifier-1	
operand-3	size-identifier-1	
operand-4	data-item-identifier-2	
operand-5	pointer-identifier-2	
operand-6	size-identifier-2	

Byte 1 is the operation code (X'68').
Operands-1,4 refer to string data items.
Operands-2,3,5,6 refer to binary data items.

MOVE *Move* **MOVE**

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

Type: Arithmetic instruction.

Function: (Operand-2) → operand-1

Description: Operand-2 is moved to operand-1. The contents of operand-2 remain unchanged.
The following mixed transfers are allowed:
Operand-1 is binary and operand-2 is decimal;
Operand-1 is decimal and operand-2 is string;
or
Operand-1 is decimal and operand-2 is binary.

These conversions are done according to the type of the receiving data item.

Condition register: Unchanged.

Rules:

Operand-1 \ Operand-2	BIN	BCD	STRG
BIN	1	3	
BCD	4	2	5
STRG		6	1

BIN→BIN
STRG→STRG

1. The content of operand-2 is moved to operand-1 from *left to right*. If the content of operand-2 is shorter than operand-1, the last character of operand-2 is repeated until operand-1 is filled. If operand-2 is longer than operand-1, the move ends when operand-1 is filled.

BCD→BCD

2. The content of operand-2 is moved to operand-1 from *right to left*. If operand-2 is shorter than operand-1, the remaining positions of operand-1 are filled by the character X'F'. If operand-2 is longer than operand-1, the move ends when operand-1 is filled. The sign is always moved to the leftmost (i.e. most significant) position.

MOVE *Continued* **MOVE**

BCD→BIN 3. The content of operand-2 is converted from decimal to binary and moved to operand-1. If the value of operand-2 is outside the range – 32768 to 32767, the result is unpredictable and overflow is indicated.

BIN→BCD 4. The content of operand-2 is converted from binary to decimal and moved to operand-1. If operand-1 is shorter than is required by the value of operand-2, the least significant digits only are moved. The sign is moved to the leftmost (i.e. most significant) position.

STRG→BCD 5. The content of operand-2 is converted from string to decimal and moved to operand-1 from *right* to *left*. Non-numeric ISO-7 characters in operand-2 are ignored (i.e. skipped). The sign of operand-1 is set negative if operand-2 contains a leading "–" sign. If operand-2 is shorter than operand-1, the remaining positions of operand-1 are filled by the character X'F'.

BCD→STRG 6. The contents of operand-2 is converted from decimal to string and moved to operand-1, from *left* to *right*. The BCD space characters, X'F' will be supressed. The sign in the decimal data item will be converted and stored in the first character position of the string data item referenced by operand-1.

X'B', plus sign is converted to '+' character
X'D', minus sign is converted to '–' character
X'O', zero is converted to ' ' character

If operand-1 is longer than operand-2, the result in the string data item is padded to the right with null characters (X'OO'). If operand-1 is shorter than operand-2 only the right most digits are moved.

Examples:

```
MOVE    FIELD1,=W'825'      FIELD1 is declared as BIN
MOVE    WORK1,INPBUF        WORK1 and INPBUF are declared as
                              BCD
MOVE    FIELD1,WORK1
MOVE    WORK1,FIELD1
MOVE    WORK1,=C'ABCDEF'
```

Intermediate code format:

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

Byte 1 is the operation code (X'00' or X'01').
Operand-1 is a reference to a data item.
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to data-item-identifier-2, array-identifier-2 or a formal parameter.

MUL *Multiply* **MUL**

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

Type: Arithmetic instruction.

Function: (Operand-1) x (Operand-2) → operand-1

Description: Operand-1 is multiplied by operand-2 and the result is placed in operand-1. The contents of operand-2 remain unchanged. A single data item may be used for both operand-1 and operand-2. In this case the data item is merely multiplied by itself. Both operands must be decimal or binary.

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

Example: MUL WORK1,INPBUF Both identifiers are declared as BCD or BIN.

Intermediate code format:

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

Byte 1 is the operation code (X'08' or X'09').
Operand-1 is a reference to a data item.
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to data item or a formal parameter.

MWAIT — *Multiple Wait* — MWAIT

Syntax: [statement-identifier] ␣ MWAIT ␣ index-identifier, data-set-identifier-1 [,data-set-identifier-2] . . .

Type: I/O instruction.

Description: This instruction is used to wait for the completion of the first of a series of events initialized with the nowait option.
After completion of one of the events, execution is continued. The binary-data-item, referenced by index-identifier, will receive the index value of the data set in the list, which has just completed its operation. First data set in the list will be referenced with index value 1.
The contidion register is set according to the status of the last operation of the relevant data set.

Condition register:
= 0 if I/O successful (OK)
= 1 if End of file (EOF)
= 2 if Error (Err)
= 3 if Begin or End of device (BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	uncond

Example: MWAIT INDX, DSKB1, DSGTP

Intermediate Code Format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	index-identifier							
byte n	list-length							
operand-2	0	0	data-set-identifier-1					
operand-3	0	0	data-set-identifier-2					

Byte 1 and 2 are filled by the system.
Byte 2 contains a reference to an external system routine.
Operand-1 is a reference to a binary data item.
byte n is filled by the CREDIT translator and contains the number of data sets present in the list.
Operands-2,3 etc. are the references to the relevant data sets.

NKI *Numeric keyboard input* **NKI**

Syntax: [statement-identifier] ␣NKI␣ [.NW,] [.NE,] data-set-identifier, data-item-identifier, key-table-identifier, size-identifier index-identifier

Type: I/O instruction.

Description: Numeric characters are read from a keyboard indicated by data-set-identifier and stored in a string data item indicated by operand-4. The number of requested characters is given in the data item specified by size-identifier, which on completion of input will contain the number of characters transferred. The data item specified by index-identifier, is filled with the position number of the terminating character in the key table. The first character of the key-table is counted as one. Key-table-identifier refers to the relevant key table. .NW and .NE indicate that the no wait and no echo options are required.

Transfer of numeric characters is ended if:

1) One of the terminating characters listed in the key table is input.
2) A character neither numeric nor listed in the key table is input.
3) The size of the string data item is reached.
4) Power failure occurs.
5) Requested number of characters is reached.
6) A key-lock switch is turned

In case 2), 3) and 5) above, the pointer will contain an undefined value and the condition register is set to ERROR.

In case of power failure the pointer is set to zero and no indication is given in the condition register.

All character positions not affected by the input are set to X'00'.

In case 6) the index value will be negative, thus indicating that a key-lock switch is turned.

The possible negative values in the index for keyboards PTS6236, 6271 and 6272 are:

−1: key-lock no.4 turned OFF
−2: key-lock no.3 turned OFF
−3: key-lock no.2 turned OFF
−4: key-lock no.1 turned OFF
−5: key-lock no.4 turned ON
−6: key-lock no.3 turned ON
−7: key-lock no.2 turned ON
−8: key-lock no.1 turned ON

If all keys are OFF, the keyboard is considered to be inactive.

NKI *Continued* **NKI**

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

Condition mask:

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

Example: NKI DSKBN,INBUF,KTAB2,INLEN,INDEX

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	E	data-set-identifier					
operand-2	data-item-identifier							
operand-3	key-table-identifier							
operand-4	size-identifier							
operand-5	pointer-identifier							

Bytes 1 and 2 are filled by the system.
Byte 2 contains a reference to an external system routine.
W is the wait bit.
W=0 no wait
W=1 wait
E is the echo bit
E=0 no echo
E=1 echo
Operand-1 is a reference to the relevant data set.
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a key table which is assumed to be literal.
Operand-4 is a reference to a binary data item.
Operand-5 is a reference to a binary data item.

PAUSE *Pause* **PAUSE**

Syntax: [statement-identifier]␣PAUSE

Type: Scheduling instruction.

Description: The execution of the task is inhibited until a restart instruction is issued by another task.

Condition register: Not significant.

Intermediate code format:

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

Bytes 1 and 2 are filled by the system. Byte 2 is a reference to an external system routine.

PERF *Perform* **PERF**

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

Type: subroutine control instruction.

Function: PP = subroutine-identifier.

Description: Control is given to a subroutine which is written in the CREDIT language. The subroutine may be in the same module as the perform instruction or in another CREDIT module.

The return address is saved on the stack. In a virtual memory system the current segment number, return address and parameter list are saved on the stack.

A maximum of eight parameters can be passed.

When an array element is passed, then the array name and index are passed each as a parameter. A format-table reference may also be passed, but not a reference to an element in the format-table. Literal parameters of the type "X" are not allowed.

Condition register: Not significant.

Example:

```
PERF␣SUB1,ARRAY,INDEX
PERF␣SUB2,DSVDU
```

Intermediate code format:

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

Byte 1 contains the operation code (X'80')
Operand-1 is a reference to a subroutine.
Operands-2,3 etc. are references to the parameters.

PERFI *Indexed perform* **PERFI**

Syntax: [statement-identifier] ␣ PERFI ␣ index-identifier, subroutine-identifier . . .

Type: Subroutine control instruction.

Description: Control is passed to the subroutine in the subroutine identifier list according to the contents of the data item specified by index-identifier.

The return address is saved on the stack. In a virtual memory system the current segment number, return address and parameter list are saved on the stack.

The first identifier in the subroutine list has the index value one. If the index is zero or greater than the number of identifiers in the subroutine list, the instruction following the indexed perform is executed.

In a system with MMU, literals, key tables and format lists can only be passed as parameter, using the PLIST directive. The number of parameters must be the same for each subroutine mentioned in this instruction. Literal parameters of the type 'X' are not allowed.

Condition Register: Not significant.

Intermediate code format:

Byte 1	0 0 1 1	0 0 1 1
operand-1	index-identifier	
Byte n	list-length	
operand-2	subroutine-identifier-1	
operand-3	subroutine-identifier-n	

Byte 1 is the operation code (X'33').
Operand-1 refers to a binary data item.
Byte n is filled by the CREDIT translator and contains the number of identifiers present in subroutine identifier list.
Operands-2,3 etc. are references to the subroutine.

PRINT PRINT **PRINT**

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

Type: Format control ; C

Description: From the current format list the line number contained in the binary data item referenced by data-item-identifier-1, up to the line number contained in the binary data-item referenced by data-item-identifier-2, is output to the data set indicated by data set identifier. The first line number on the output device is always one. The contents of the data-item (second line number) referenced by data-item-identifier-2 may be equal or greater than the first line number contained in the data-item referenced by data-item-identifier-1.
When the second line number is zero all the lines from the current format list are output to the data set from the first line number up to the last one. Two spaces, which serve as control character (i.e. line feed and carriage return), are always output as the last line. The second line number may be indicated by a literal of the type binary.

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

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	UNCON-DITIONAL

Example:
PRINT ␣ DSGP, LINE 5, LINE 15
PRINT ␣ DSGP, LINE 5, = '0'

Intermediate code format:

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

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

READ *Read* **READ**

Syntax: [statement-identifier] READ [.NW,] [.NEA,] data-set-identifier data-item-identifier, size-identifier

Type: I/O instruction

Description: Characters are read from the device indicated by data-set-identifier into a string data item indicated by operand-2.
.NW indicates that the no wait option is required.
The transfer of characters is ended if:

1) An end-of-record condition is encountered
2) The string size is reached

The number of characters which are transferred, is returned by the system in the data item specified by size-identifier.
If the data item size is exceeded, error is set in the condition register.

The disk sequential access method is as follows.
The record to be read depends on the last data management function called by this task. If no data management function has been called by this task for this file (indicated by data set identifier) after the file was assigned, the current record number (CRN) will point to the first record of the file to be read. The logical record number can be fetched with GET CURRENCY (DSC1). If the reading is successful, the record is set under exclusive access for this task.
.NEA option indicates that exclusive access should not be set for this record.
When the data-set-identifier refers to a data communication data set this instruction will read data from the line. Time out must be set before this instruction is executed, with the DSC1 instruction and control value X'0B'. In a DC task time out must be set to zero.

For intertask communication data-set-identifier refers to a data set in which the input file code is defined. When a READ (unaddressed) is issued by a task, first a check is performed on the queue of the task that issues the READ for RWRITE (addressed). If an addressed write is in the queue for this task, the instruction is completed. When no match occurs the WRITE (unaddressed) queue is checked, if not empty the first one is removed from the queue and the instruction is completed. Else the instruction is put into the queue for unaddressed read.

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

Condition mask:

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

Example:
```
READ    DSKBN, BUF1, SIZE
READ    .NEA, DSDK1, BUF1, SIZE
```

READ *Continued* **READ**

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	E	data-set-identifier					
operand-2	data-item-identifier							
operand-3	size-identifier							

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine.
W is the wait bit
W=0 no wait
W=1 wait
E is the echo bit
E=1 echo
E=0 no echo
Operand-1 is the reference to the relevant data set
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a binary data item.

RET *Return* **RET**

Syntax: [statement-identifier] ␣RET␣ [equate-identifier / decimal-integer]

Type: Subroutine control instruction.

Description: Control is passed back to the calling module and execution is continued at the instruction following the original perform or indexed perform instruction.

The return address is found on the stack. In a virtual memory system the proper segment number and return address are found on the stack.

Decimal-integer specifies a displacement (number of bytes) which has to be added to the normal return address. This displacement excludes the length of the parameter list, in bytes, which may have been passed to the subroutine.

Condition register: Not significant.

Example:
```
RET
RET 2
```

Intermediate code format:

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

Byte 1 is the operation code (X'34').
Byte 2 contains displacement.

RREAD *Random Read* **RREAD**

Syntax: [statement-identifier] ␣ RREAD ␣ [.NW,] [.NEA,]
data-set-identifier, data-item-identifier-1,
size-identifier, data-item-identifier-2

Type: I/O instruction.

Description: A record is read from the file indicated by data-set-identifier and stored in a string data item indicated by data-item-identifier-1.
The number of requested characters is given in the data-item specified by size-identifier, which on completion of input will contain the number of characters transferred.
The logical record number is given in a binary or decimal data item indicated by data-item-identifier-2.
.NW indicates that no wait option is required.
.NEA option indicates that exclusive access should not be set for this record.
On a successful read, the accessed record is available for this task under exclusive access. (Not accessible by other tasks).
The current record number (CRN) will point to the current data record.
Exclusive access is automatically released after:
- a write of the record
- a delete function.

The exclusive access may be released explicity by the "Release exclusive access" function.
For intertask communication data-set-identifier refers to a data set in which the input file code is defined.
Data-item-identifier-2 refers to a binary data item, which contains the task identifier of the addressed task. If the addressed task has not issued a RWRITE (to this task) or WRITE, then this request will be queued on the addressed task. In the other case the instruction will be completed.

Condition register:
= 0 if I/O successful (OK)
= 1 if End of File (EOF)
= 2 if Error (ERR)
= 3 if Begin or End of device (BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	Unconditional

Example: RREAD DSDK, BUFRC, LENGTH, RECNR

RREAD *Continued* **RREAD**

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	EA	data-set-identifier					
operand-2	data-item-identifier-1							
operand-3	size-identifier							
operand-4	data-item-identifier-2							

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine.
W is the wait bit. EA is the exclusive access bit.
W=0 no wait EA=0 exclusive access.
W=1 wait EA=1 no exclusive access.
Operand-1 is the reference to the relevant data set.
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a binary data item.
Operand-4 is a reference to a binary or decimal data item.

RSTRT *Restart* **RSTRT**

Syntax: [statement-identifier] ␣RSTRT␣ task-identifier

Type: Scheduling instruction.

Description: The task in pause mode indicated by task-identifier is restarted. Task-identifier is a reference to a binary or string data item. In case of a string data item the first two bytes must contain the task identity.

Condition register: Not significant.

Intermediate code format:

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

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

RWRITE *Random Write* **RWRITE**

Syntax: [statement-identifier] ␣ RWRITE ␣ [.NW,]
data-set-identifier, data-item-identifier-1, data-item-identifier-2

Type: I/O instruction

Description: The record present in a string data item, indicated by data-item-identifier-1 is written to the file indicated by data-set-identifier. Before it is written the status of the record is checked whether it is "FREE" or "USED".
When "FREE" the status is changed to "USED" and the record is written.
When the status is "USED" the record will be written only if it is under Exclusive access for this task.
If it is not under exclusive access the error "record protected" will be sent.
The logical record number, for disc file, is in the binary or decimal data item indicated by data-item-identifier-2. After a random write, exclusive access is released. Random Write may be used to write on a display, data-item-identifier-2 refers in this case to a binary-data-item which contains the cursor position, where writing start.
For intertask communication, data-set-identifier refers to a data set in which the output file code is defined.
Data-item-identifier-2 refers to a binary data item, which contains the task identifier of the addressed task. If the addressed task has not issued a RREAD (to this task) or READ then this request will be queued on the addressed task. In the other case the instruction will be complete →

Condition register:
= 0 if I/O successful (OK)
= 1 if End of File (EOF)
= 2 if Error (ERR)
= 3 if Begin or End of Device (BEOD)

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	$\overline{OK}$	$\overline{EOF}$	$\overline{ERR}$	Unconditional

Example: RWRITE DSDK, BUFRC, RECNR

Intermediate code format:

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

RWRITE *Continued* **RWRITE**

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.
W is the wait bit
W=0 no wait
W=1 wait
Operand-1 is the reference to the relevant data set.
10/100 refers to the first data set.
Operand-2 is a reference to a string data item.
Operand-3 is a reference to a binary or decimal data item.

SB *Short branch* **SB**

Syntax: [statement-identifier]␣SB␣[{ equated-identifier, / condition-mask, }] statement-identifier

Type: Branch instruction.

Description: The instruction to be executed is indicated by statement-identifier, if operand-1 matches the contents of the condition register. Otherwise the instruction following the short branch will be executed. If operand-1 is omitted an unconditional branch (value 7) is generated.

Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the short branch (incl. 2 bytes of the short branch), or 255 bytes after the short branch.

Condition register: Not changed.

Example:
```
SB    INP3
SB    2,INP4
```

Intermediate code format:

Byte 1	0	1	0	1	B	CND	
Byte 2	displacement						

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

SET *Set* **SET**

Syntax: [statement-identifier]␣SET␣ data-item-identifier

Type: Logical instruction.

Function: 1 → data-item-identifier

Description: The content of data-item-identifier is set to one. (TRUE)
Data-item-identifier must refer to a boolean data item. (length 1 bit)
The condition register is set according to the *previous* value of the content of data-item-identifier.

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

Condition mask:

0	1	2	3	4	5	6	7
DI=0	–	–	–	DI≠0	–	–	–

Intermediate code format:

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

Byte 1 is the operation code (X'41').
Operand-1 is a reference to a boolean data item.

SETCUR *Set Cursor* **SETCUR**

Syntax: [statement-identifier] ␣ SETCUR

Type: Format control I/O

Description: The cursor will be positioned at the first character position of the current input field.

Condition register: = 0 if cursor positioned correctly
= 2 if I/O error

Condition mask:

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

Example: SETCUR

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 is a reference to an external system routine.

SETTIME *Set Clock* **SETTIME**

Syntax: [statement-identifier] ␣ SETTIME ␣ data-item-identifier

Type: Clock control.

Description: The system clock is set to the time specified in a string data item, indicated by data-item-identifier.
The string data item must have a length of six characters, in which is specified H H M M S S

H = hour
M = minute
S = second

The system clock is updated by the real time clock thus giving correct time of day.

Condition register: Unchanged.

Example: SETTIME, TIME

Intermediate code format:

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

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

SUB *Subtract* **SUB**

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

Type: Arithmetic instruction.

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

Description: Operand-2 is subtracted from operand-1 and the result is placed in operand-1.
Operand-2 is unchanged. Both operands must be binary or both operands must be decimal. The condition register is set according to the content of operand-1.

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

Condition mask:

0	1	2	3	4	5	6	7
=0	>0	<0	over flow	≠0	⩽0	⩾0	uncondi-tional

Example:
```
SUB   FIELD1,FIELD2     FIELD1 and FIELD2 are declared as BIN.
SUB   WORK1,=D'4317'    WORK1 is declared as BCD.
```

Intermediate code format:

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

Byte 1 is the operation code (X'04' or X'05').
L=1 operand-2 is a reference to a literal constant.
L=0 operand-2 is a reference to data-item-identifier-2, array-identifier-2 or a formal parameter.
Operand-1 is reference to a binary or decimal data item.

SWITCH *Switch task on same level* **SWITCH**

Syntax: [statement-identifier] ␣ SWITCH

Type: Scheduling instruction.

Description: The running task will be interrupted and queued last in the dispatcher queue. Control is given to another task on the same level, which is the first one in the dispatcher queue.

Condition register: Unchanged.

Example: SWITCH

Intermediate code format:

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

Byte 1 contains the operation code (X'30')
Byte 2 is a reference to an external system routine.

TB *Test and branch* **TB**

Syntax: [statement-identifier]␣TB␣{equate-identifier / condition-mask}, data-item-identifier, statement-identifier

Type: Branch instruction.

Description: The content of data-item-identifier is compared with zero and the condition register is set according to the result of this comparison. If operand-1 matches the condition register, the next instruction to be executed is found at the address specified by statement-identifier. If operand-1 does not match the condition register, the instruction following the test and branch will be executed.

Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the test and branch (incl. 3 bytes of the test and branch) or 255 bytes after the test and branch.

Data-item-identifier refers to a boolean data item.

Example: TB FALSE,LKMX1,TTGO

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

Condition mask:

0	1	2	3	4	5	6	7
DI=0	–	–	–	DI≠0	–	–	–

Intermediate code format:

Byte 1	0 1 0 0	1 0	B	C
operand-1	data-item-identifier			
Byte n	statement-identifier			

Byte 1 is the operation code (X'48' up to X'4B').
B=0 forward branching.
B=1 backward branching.
C=0 condition mask is zero.
C=1 condition mask is four.
Operand-1 is a reference to a boolean data item.
Byte n contains a displacement.

TBF *Test and branch on false* **TBF**

Syntax: [statement-identifier] ␣ TBF ␣ data-item-identifier, statement-identifier

Type: Branch instruction.

Description: The content of data-item-identifier is compared with zero and the condition register is set according to the result of this comparision. If the content of operand-1 is zero, the next instruction to be executed is found at the address specified by statement-identifier. If the content of operand-1 is one, the instruction following the test and branch on false will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the test and branch on false (incl. 3 bytes of the test and branch on false) or 255 bytes after the test and branch on false.
Data-item-identifier refers to a boolean data item.

Example: TBF LKMX1, TTGO

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

Intermediate code format:

Byte 1	0	1	0	0	1	0	B	0
Operand-1	data-item-identifier							
Byte n	statement-identifier							

Byte 1 is the operation code (X'48' or X'4A').
B = 0 forward branching
B = 1 backward branching
Operand-1 is a reference to a boolean data item.
Byte n contains a displacement.

TBT *Test and branch on true* **TBT**

Syntax: [statement-identifier] ␣ TBT ␣ data-item-identifier, statement-identifier

Type: Branch instruction.

Description: The contents of data-item-identifier is compared with zero and the condition register is set according to the result of this comparision. If the contents of operand-1 is one, the next instruction to be executed is found at the address specified by statement-identifier. If the contents of operand-1 is zero, the instruction following the test and branch on true will be executed.
Statement-identifier may only refer to a statement which is within the limit of 255 bytes before the test and branch on true (incl. 3 bytes of the test and branch on true) or 255 bytes after the test and branch on true.
Data-item-identifier refers to a boolean data item.

Example: TBT LKMX1, TTGO

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

Intermediate code format:

Byte 1	0	1	0	0	1	0	B	1
operand-1	data-item-identifier							
Byte n	statement-identifier							

Byte 1 is the operation code (X'49' or X'4B').
B = 0 forward branching
B = 1 backward branching
Operand-1 is a reference to a boolean data item.
Byte n contains a displacement.

TBWD *Tabulate backward* **TBWD**

Syntax: [statement-identifier] ␣ TBWD

Type: Format control I/O

Description: Tabulation backward from the current input field.
The current input-field number, according to the FKI-input field numbering, is decreased with one and this new input-field (current input field-1), of the current format list is made current, also when CTAB option was specified for this new current input field.

Condition register:
= 0 Operation successful.
Cursor is set to the first position of the new current input field
= 1 operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field.
= 2 Addressed input field not found in current format.
Cursor remains in its old position
= 3 An empty compulsory field was found before this instruction was executed
The compulsory field stays current and cursor remains in its old position

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	$\overline{\text{SUCC}}$	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.

TDOWN *Tabulate Down* **TDOWN**

Syntax: [statement-identifier] _ TDOWN

Type: Format control I/O

Description: Tabulation to the nearest FKI-input field on the next line. The FKI-input field on the line immediately following the current line, with a starting column nearest to the starting column of the current input field, becomes current. When the two nearest columns are found on the next line, the left FKI-input field will become current. If no FKI-input field is found on the next line, the following lines are searched in sequence.

Condition register:
= 0 Operation successful.
Cursor is set to the first position of the new current input field
= 1 operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field.
= 2 Addressed input field not found in current format
Cursor remains in its old position.
= 3 An empty compulsory field was found before this instruction was executed.
The compulsory field stays current and cursor remains in its old position.
(Not relevant for THOME).

Example:

LINE 2 12 ① 20 ② 40 ③

LINE 4 12 ④ 22 ⑤ 40 ⑥

FKI-input field number 2, starting in column 20 is current. TDOWN results now in FKI-input field number 5 becoming current.

Condtion mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	SUCC	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Byte 1 ... by the system.
Byte 2 ... external system routine.

TEST *Test* **TEST**

Syntax: [statement-identifier] ␣TEST␣ data-item-identifier

Type: Logical instruction.

Function: (data-item-identifier) ← 0

Description: The content of data-item-identifier is compared with zero and the condition register is set according to the result of this comparison.

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

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

Condition mask:

0	1	2	3	4	5	6	7
DI=0	–	–	–	DI≠0	–	–	–

Intermediate code format:

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

Byte 1 is the operation code (X'43').
Operand-1 is a reference to a boolean data item.

TESTIO *Test I/O completion* **TESTIO**

Syntax: [statement-identifier] ␣ TESTIO ␣ data-set-identifier.

Type: I/O instruction.

Description: The data set indicated by data-set-identifier is tested for completion of the I/O. (without wait).

Condition register: = 0 if I/O is completed (OK)
= 1 if I/O is not completed (EOF)

Condition mask:

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

Example: TESTIO DSVO

Intermediate object code format:

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

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

TFWD *Tabulate forward* **TFWD**

Syntax: [statement-identifier] ␣ TFWD

Type: Format control I/O

Description: Tabulation forward from the current input field.
The current input field number, according to the FKI-input field numbering, is increased with one and this new input field (current field number + 1) of the current format list is made current.

Condition register:
= 0 Operation successful
Cursor is set to the first position of the new current input field
= 1 operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field.
= 2 Addressed input field not found in current format.
Cursor remains in its old position
= 3 An empty compulsory field was found before this instruction was executed
The compulsory field stays current and cursor remains in its old position

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	$\overline{\text{SUCC}}$	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.

THOME — *Tabulate Home* — THOME

Syntax: [statement-identifier] ␣ THOME

Type: Format control I/O

Description: The first FKI-input field of the current format list is made current.

Condition register:
= 0 Operation successful
Cursor is set of first position of the new current input field.
= 1 Operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field
= 1 Operation successful.
Cursor remains in its old position, because the CTAB flag is set for this current input field.
= 2 Not relevant
= 3 Not relevant

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	$\overline{\text{SUCC}}$	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.

TLDOWN *Tabulate left down* **TLDOWN**

Syntax: [statement-identifier] ␣ TLDOWN

Type: Format control I/O

Description: Tabulation to the left-most FKI-input field on the following line. The first FKI-input field on the line immediately following the current one becomes current.
If no FKI-input field is found on that line, the following lines are searched in sequence.

Condition register:
= 0 Operation successful.
Cursor is set of first position of the new current input field.
= 1 Operation successful.
Cursor remains in its old position, because the CTAB flag is set for this current input field.
= 2 Addressed input field not found in current format.
Cursor remains in its old position
= 3 An empty compulsory field was found before this instruction was executed.
The compulsory field remains current and the cursor remains in its old position.
(Not relevant for THOME)

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	$\overline{\text{SUCC}}$	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 is a reference to an external system routine.

TLEFT — *Tabulate left* — TLEFT

Syntax: [statement-identifier] ␣ TLEFT

Type: Format control I/O

Description: Tabulation to the left-most input field on the current line.
The left-most FKI-input field on the same line as the current FKI-input field, becomes current.
Note: This input field always exists.

Contion register:
= 0 Operation successful
Cursor is set to the first position of the new current input field
= 1 Operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field
= 2 Not relevant
= 3 An empty compulsory field was found before this instruction was executed.
The compulsory field stays current and cursor remains in its old position.

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	$\overline{\text{SUCC}}$	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are filled by the system.
Byte 2 is a reference to an external system routine.

TRIGHT *Tabulate right* **TRIGHT**

Syntax: [statement-identifier] ⊔ TRIGHT

Type: Format control I/O

Description: Tabulation to the right-most input field on the current line.
The right-most FKI-input field on the same line as the current FKI-input field, becomes current.
Note: This input field always exists.

Condition register:
= 0 Operation successful
Cursor is set to the first position of the new current input field
= 1 Operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field
= 2 Not relevant
= 3 An empty compulsory field was found before this instruction was executed.
The compulsory field stays current and cursor remains in its old position.

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	SUCC	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are fully filled by the system
Byte 2 is a reference to an external system routine.

TSTCTL *Test control flag* **TSTCTL**

Syntax: [statement-identifier] ␣ TSTCTL ␣ control value

Type: Format control I/O

Description: One of the control flags, of the current input field is tested and the condition register is set according to the result.
The control flags are specified in a FKI-format list declaration.
Control value specifies which flag has to be tested.

Control value	Significance
0	Test "ALPHA" flag
1	Test "REWRT" flag
2	Test "ME" flag
3	Test "NEOI" flag
4	Test "NCLR" flag
5	Test "CTAB" flag
6	Test "VERIF" flag.

Condition register: = 0 when a flag is not set.

Condition mask:

0	1	2	3	4	5	6	7
NOT SET	–	–	–	SET	–	–	UNCONDITIONAL

Example: TSTCTL 2

Intermediate object code:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	control value							

Bytes 1 and 2 are filled by the system
Byte 2 is a reference to an external system routine.
Operand-1 is the control value.

TUP *Tabulate Up* **TUP**

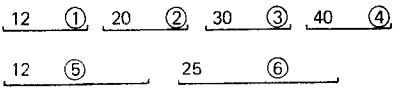
Syntax: [statement-identifier] ␣ TUP

Type: Format control I/O

Description: Tabulation to the nearest FKI-input field on the preceding line. The FKI-input field on the line immediately preceding the current line, with a starting column nearest to the starting column of the current input field, becomes current. When the two nearest columns are found on the preceding line, the left FKI-input field will become current. If no FKI-input field is found on the preceding line, the line preceding that one is searched etc.

Condition register:
= 0 Operation successful
Cursor is set to the first position of the new current input field.
= 1 Operation successful
Cursor remains in its old position, because the CTAB flag is set for this current input field.
= 2 Addressed input field not found in current format.
Cursor remains in its old position.
= 3 An empty compulsory field was found before this instruction was executed. The compulsory field stays current and cursor remains in its old position.
(Not relevant for THOME).

Example:



FKI-input field number 6, starting in column 25 is current.
TUP results now in FKI-input field number 2 becoming current.

Condition mask:

0	1	2	3	4	5	6	7
SUCC	SUCC CTAB	NOT FOUND	EMPTY COMP. FIELD	$\overline{\text{SUCC}}$	NO SUCC CTAB	FOUND	UNCON-DITIONAL

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 is a reference to an external system routine.

UNUSE *unuse* **UNUSE**

Syntax: [statement-identifier] ␣ UNUSE ␣ block-identifier

Type: Storage control instruction.

Description: The user workblock or swappable workblock, attached to the current task will be detached. A swappable workblock is restored on disk.

The condition register is unequal to zero if the referenced workblock does not exist.

Condition register:
= 0 workblock correctly detached from the task.
= 2 no workblock of this type was attached to the task.

Intermediate object code:

Byte 1	0 0 1 1 0 0 0 0
Byte 2	external reference
operand-1	UWB/SWB type
Byte n	Block number

Bytes 1 and 2 are filled by the system.
Operand-1 is type of user or swappable workblock.
Byte n is the index to the user or swappable workblock.

UPDFLD *Update Input Field* **UPDFLD**

Syntax: [statement-identifier] UPDFLD ␣ control value, data-item-identifier

Type: Format control I/O

Description: The contents of the string data item referenced by data-item-identifier (buffer), is moved to the data-item of the current input field according to the rules as valid for the MOVE-instruction. Depending on control value the new contents of the deta-item is redisplayed on the screen.

Control value	Significance
0	Redisplay only if for the current input field the "REWRT" flag is set.
1	Redisplay always.

Condition register: = 0 if OK
= 2 if I/O error.

Condition mask:

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

Example: UPDFLD 1, SPINPUT

Intermediate object code:

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

Bytes 1 and 2 are filled by the system
Byte 2 is a reference to an external system routine
Operand-1 is the control value
Operand-2 is a reference to a string data item.

USE *Use* **USE**

Syntax: [statement-identifier] ␣ USE ␣ block-identifier, data-item-identifier

Type: Storage control instruction.

Description: After the USE instruction, the task may access the user or swappable work block specified by block-identifier and data-item-identifier, which must be binary. If the data-item does not specify an existing user or swappable workblock, i.e. it contains a number higher than the highest numbered user or swappable work block, the condition register will be set unequal to zero.
A user or swappable workblock is released from the task as a result of a UNUSE instruction specifying the same user or swappable work block type.

Example: USE UB1, BLOCKNO

Condition register: =0 if index value within permitted limits.
=1 if swappable workblock under exclusive access.
=2 if index value out of permitted limits.

Intermediate code format:

Byte 1	0 0 1 1 0 0 0 0
Byte 2	external reference
operand-1	UWB/SWB type
Byte n	block number
operand-2	data-item-identifier

Bytes 1 and 2 are filled by the system.
Operand-1 is the type of user or swappable workblock.
Byte n is the index to the user workblock or swappable workblock.
Operand-2 is a reference to a binary data item.

WAIT *Wait* **WAIT**

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

Type: I/O instruction

Description: If the most recently started operation on the data set indicated by data-set-identifier is not yet completed, execution of the next instruction is inhibited. After completion of the operation, execution is continued.

If the operation is already completed before this instruction is executed, the program continues as normal without taking any action on this instruction.

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

Condition mask:

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

Example: WAIT DSKBN

Intermediate code format:

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

Bytes 1 and 2 are filled by the system. Byte 2 contains a reference to an external system routine.
W is the wait bit. This has no significance for WAIT.
Operand-1 is a reference to the relevant data set.
10/100 refers to the first data set.

WRITE *Write* **WRITE**

Syntax: [statement-identifier] ␣ WRITE ␣ [.NW,] data-set-identifier, data-item-identifier [,size-identifier]

Type: I/O instruction.

Description: The contents of the string data item, indicated by operand-2 is output to the devices specified by data-set-identifier.

The number of characters to be output can be given in the data item specified by size-identifier, which on completion of the output will contain the number of characters transferred.

When the data-set-identifier refers to a data communication data set, this instruction will send data on the line. Time out must be set before this instruction is executed, with the DSC1 instruction and control value X'0B'.

For intertask communication data-set-identifier refers to a data set in which the output file code is defined. When a WRITE (unaddressed) is issued by a task, first a check is performed on the queue for RREAD (addressed). If an addressed read is in the queue for this task, the instruction is completed. When no match occurs the READ queue (addressed) is checked, if not empty the first one is removed from the queue and the instruction is completed. Else the instruction is put into the queue for unaddressed write.

.NW indicates that no wait is desired.

The first two bytes in the string data item must contain a control character. The first byte must always be unequal to zero and the *second byte* must contain the control character (Except for disk). The function of the control character is device dependent. The control character may have the following value:

- General Terminal Printer or Line Printer
  X'2B': print the line without advancing the paper.
  X'30': advance two lines before printing.
  X'31': skip to top of form before printing. (only for line printer)
  Other codes: one line feed is executed before printing.
  Special characters allowed in the user buffer and not restricted to the first word in the buffer:
  X'11' Tabulation character. This character should be followed by two ISO-7 digit characters giving the tabulation position. (Only for GTP).
- Teller Terminal Printer PTS6222, PTS6223
  Voucher/passbook printing
  X'2B': print the line without advancing the paper.
  X'30': advance two line steps before printing.
  X'31'–X'39': advance paper 1–9 line steps before printing.
  Other codes: one line step is executed before printing.

  Journal/tally roll printing
  X'30': advance two step lines before printing. (Two steps = one line feed).

Other codes: one line step is executed before printing.
Special characters allowed in the user buffer:
X'09': The print head is moved to the right most print position of the voucher. This character should be present in the last buffer position.
X'0D': The print head is moved to the right most position of the journal station. This character should be present in the last buffer position.

- Teller Terminal Printer (PTS6371)
  The control character present in the second character of the buffer, as follows:
  /28 – printing is carried out from the last position of the previously printed line on this device. However, if the character pitch has been set, or if positioning has been carried out to the same line, since the previous line was printed, the printing will be from the tabulation position on the present line.
  /30 – the paper is advanced two lines, and the printing carried out from the tabulation position.
  /31 – journal: the paper is advanced three lines and the printing carried out from the tabulation position. This will make the previously written data readable through the window on the journal station.
  – document: printing is started from the tabulation position on line 1.
  Any other value in the control code will cause one line feed before printing from the tabulation position.
  The requested length must include the two bytes used for the control code, but if it is two, only the action specified by the control code, is carried out.
  The maximum line length on the two print stations is limited to the following, based on normal character width:

	Journal	Document
10 characters/inch	33	80
12 characters/inch	40	96
15 characters/inch	50	120

  One expanded character equals two normal characters.

- Numeric and Signal Display
  X'30'–X'3F': these codes are sent to the first position (i.e. the left most program display tube on the indicator unit).
  X'40'–X'4F': these codes are sent to the second position.
  X'50'–X'5F': these codes are sent to the third position.
  X'60'–X'6F': these codes are sent to the fourth position.

WRITE *Continued* WRITE

- Disk, sequential access method
  Data-item-identifier indicates the string to be written to the disk file, indicated by the data-set-identifier. The record will be written, directly after the Last-Record-Number (LRN), which is administrated by Data Management. The record status (FREE or USED) is checked by Data Management before the record is written.

- Video display or plasma display
  X'2B': The text is displayed from current cursor position.
  X'30': Cursor is advanced two lines and positioned at the beginning of the line, before the text is displayed.
  X'31': Erase display and position cursor on home position before the text is displayed.
  Other codes: Advance cursor one line before the text is displayed.
  Special characters allowed in the user buffer and not restricted to the first word in the buffer:

  Characters Valid for All Displays
  /AE: Displayed as point (/E2)
  /11: Tabulation character. This character should be followed by two ISO-7 digits giving the tabulation position.
  /07: Bell is sent to the display

  Characters Valid for PTS 6344 only
  /12: Underline start. Output of characters which follow this character are provided with underline
  /13 Underline stop. Output of characters which follow after this character are *not* provided with underline. Underline stop mode will also appear at request end
  /14 Fast output. First character following /14 will be transmitted in fast output mode up to requested length. Note that cursor will remain unchanged.
  /1C: Data to keyboard.
  /1D: Master clear to keyboard.
  /1E: Low intensity start. Output of characters which follow after this character, are displayed at low intensity.
  /1F: Low intensity stop. Output of characters which follow after this character are displayed at normal intensity. Normal intensity mode will also appear at request end.

  See EDWRT for special characters in buffer for PTS6371.

Condition register :
= 0 if I/O successful (OK)
= 1 if End of File (EOF)
= 2 if Error (ERR)
= 3 if Begin or End of device (BEOD)

WRITE *Continued* **WRITE**

Condition mask:

0	1	2	3	4	5	6	7
OK	EOF	ERR	BEOD	OK	EOF	ERR	Unconditional

Example: WRITE DSTJT, OUTSTR

Intermediate code format:

Byte 1	0	0	1	1	0	0	0	0
Byte 2	external reference							
operand-1	W	data-set-identifier						
Byte n	list-length							
operand-2	data-item-identifier							
operand-3	size-identifier							

Bytes 1 and 2 are filled by the system.
Byte 2 contains a reference to an external system routine.
W is the wait bit.
W=0 no wait
W=1 wait
Operand-1 is a reference to the relevant data set.
10/100 refers to the first data set.
Byte n is filled by the translator.
Operand-2 contains a reference to a string data item.
Operand-3 contains a reference to a binary data item.

XCOPY *Extended Copy* **XCOPY**

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

Type : String instruction

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

Description: Starting at pointer-identifier-2, the content of operand-4 is copied from left to right to operand-1 beginning at pointer-identifier-1.
The number of bytes to be copied is specified by size-identifier.
This XCOPY is possible between two decimal data items, two binary data items or two string data items. Also copying between two data items of different type is allowed, without conversion.
The first characters of operand-1 and operand-4 are counted as zero when setting the pointer.

Condition register : unchanged

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

Intermediate code format :

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

Byte 1 is the operation code (X'6A')
operands-1,4 are references to string data items or decimal data items.
operands-2,3,5 are references to binary data items.

XSTAT *Extended status transfer call* **XSTAT**

Syntax: [statement-itentifier] ␣ XSTAT ␣ data-set-identifier, data-item-identifier

Type: I/O instruction.

Description: A 16 bit device dependent status code from the data set indicated by data-set-identifier, is transferred to the binary data item indicated by operand-2.

Extended status codes are explained in appendix 2.

Condition register: Unchanged

Example: XSTAT DSCASS,STATUS

Intermediate code format:

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

Bytes 1 and 2 are filled by the system. Byte 2 contains a reference to an external system routine.
W is the wait bit. This has no significance for XSTAT.
Operand-1 is a reference to the relevant data set.
10/100 refers to the first data set.
Operand-2 is a reference to a binary data item.

1.4.9 *Declaration Reference*

This section describes the syntax and use of each declaration. The possible values of the variables in declarations is given in appendix 1. The notation conventions are described in section 1.1.5.

CON *Constant* **CON**

Syntax: [identifier] ␣ CON ␣ { actual parameter / value / value expression } [,actual-parameter / ,value / ,value expression] ...

Type: Parameter declaration.

Description: This declaration specifies parameter(s), which have to be passed to a subroutine. The CON declaration follows immediately after a PERF, PERFI or CALL instruction.

Note: It is recommended to use the PLIST declaration instead.

Format branch on condition

Syntax:

[identifier] ␣ { FBN / FBNN / FBNP / FBNZ / FBP / FBZ } ␣ data-item-identifier, identifier

Type: Format list declaration

Description: These instructions perform forward branching only.
See specific format branch on condition, reference.

FB *Format Branch* **FB**

Syntax: identifier ␣ FB ␣ identifier

Type: Format-list-declaration.

Description: Editing continues at the format-list-declaration referenced by identifier, further down in the same format list. When the identifier refers to the FMEND declaration, editing is terminated.

FBF
FBT

Format Branch on false/true

FBF
FBT

Syntax: [identifier] ␣ { FBF / FBT } ␣ data-item-identifier-1, identifier.

Type: Format-list-declaration

Description: If the value of the boolean data item referred to by data-item-identifier is TRUE, the FBF will result in continuation of the editing at the declaration following the FBF. When the contents of the boolean data-item is FALSE, then editing will be continued at the format list declaration referred by identifier.
If the boolean data item is TRUE, the FBT will result in continuation of editing at the format list declaration referred by identifier, when the boolean data item is FALSE, the next format list declaration will be executed.
Branching to the FMEND results in termination of editing.
The contents of the data item may not be changed while the format list concerned is current.

FBN *Format Branch on Negative* **FBN**

Syntax: [identifier] ␣ FBN ␣ data-item-identifier, identifier.

Type: Format-list-declaration.

Description: If the contents of the binary or decimal data-item, referred by data-item-identifier, is negative, editing continues at the format-declaration indicated by identifier. (Forward branching only). If the contents of the binary or decimal data-item is positive or zero editing continues at the next format-list-declaration Branching to the FMEND results in termination of editing. The contents of the data item may not be changed while the format list concerned is current.

FBNN *Format branch on not negative* **FBNN**

Syntax: [identifier]␣FBNN␣data-item identifier, identifier.

Type: Format-list-declaration.

Description: If the contents of the binary or decimal data-item, referred by data-item-identifier, is not negative or zero, editing continues at the format-list declaration indicated by identifier. (Forward branching only).

If the contents of the binary or decimal data-item is negative editing continues at the next format list declaration.

Branching to the FMEND results in terminating of editing. The contents of the data item may not be changed while the format list concerned is current.

FBNP *Format branch on not positive* **FBNP**

Syntax: [identifier] ␣ FBNP ␣ data-item-identifier, identifier

Type: Format-list-declaration.

Description: If the contents of the binary or decimal data-item, referred by data-item-identifier, is not positive or zero, editing continues at the format list declaration indicated by identifier. (Forward branching only).
If the contents of the binary or decimal data-item is positive, editing continues at the next format-list-declaration.
Branching to the FMEND results in terminating of editing.
The contents of the data item may not be changed while the format list concerned is current.

FBNZ *Format branch on not zero* **FBNZ**

Syntax: [identifier] ␣ FBNZ ␣ data-item-identifier, identifier

Type: Format-list-declaration.

Description: If the contents of the binary or decimal data item, referred by data-item-identifier, is not zero editing continues at the format-list-declaration indicated by identifier. (Forward branching only).
If the contents of the binary or decimal data item is zero, editing continues at the next format list declaration. Branching to the FMEND results in terminating of editing.
The contents of the data item may not be changed while the format list concerned is current.

FBP *Format Branch on positive* **FBP**

Syntax: [identifier] ␣ FBP ␣ data-item-identifier, identifier.

Type: Format-list-declaration.

Description: If the contents of the binary or decimal data-item referred by data-item-identifier, is positive editing continues at the format declaration indicated by identifier. (Forward branching only).
If the contents of the binary or decimal data-item is not positive or zero editing continues at the next format-list-declaration.
Branching to the FMEND results in terminating of editing.
The contents of the data item may not be changed while the format list concerned is current.

FBZ *Format branch on zero* **FBZ**

Syntax: [identifier] ␣ FBZ ␣ data-item-identifier, identifier.

Type: Format-list-declaration.

Description: If the contents of the binary or decimal data-item, referred by data-item-identifier is zero, editing continues at the format-list-declaration indicated by identifier. (Forward branching only). If the contents of the binary or decimal data-item is not zero, editing continues at the next format-list declaration. Branching to the FMEND results in terminating of editing.
The contents of the data item may not be changed while the format list concerned is current.

FCOPY *Format copy* **FCOPY**

Syntax: [identifier] ␣ FCOPY ␣ { data-item-identifier / literal }

Type: Format list declaration.

Description: The location, addressed by operand is copied into the output buffer. The data-item or literal must be of the value type string. Null characters (X'00') are not copied into the output buffer. A literal is not allowed when the FCOPY is used in conjunction with one of the input field declarations FINP or FKI.

Example:

```
FCOPY ␣ = C'BOOK'
FCOPY ␣ = X'3031'
FCOPY ␣ = X'41004243'      Result in buffer X'414243'
FCOPY ␣ = C'AB'

FCOPY  DATAID
FCOPY  DATARR(IND)         The contents of the data item specified is copied into the output buffer.
```

FCW — *Format control word* — FCW

Syntax: [identifier] ␣FCW ␣{data-item-identifier / literal constant}

Type: Format list declaration

Description: The operand must refer to a data-item or literal of the type binary. If the control word for a particular device must be zero, then, in an EDWRT instruction, FCW causes end of record, while in an EDIT instruction the editing is terminated.

Example:
```
FCW   BIN1
FCW␣= '0'
FCW␣= X'8000'
```

FEOR *Format end of record* **FEOR**

Syntax: [Identifier] FEOR

Type: Format list declaration

Description: If the format list is used by an EDWRT or DISPLAY instruction the edited buffer is output to the data-set specified in the instruction. No output request is performed, when the buffer is empty.

After an EDWRT instruction the condition register will, if there is more than one FEOR in the format list, contain the logical sum of the conditions met at each output.

If used in an EDIT instruction the FEOR statement causes termination of the editing.

Example:

```
FRMT
FILLR   ' ',2
FTEXT   'ALPHANUMERIC TEXT'
FTAB    16
FTEXT   'STRING'
FEOR
FILLR   ' ',2
FTEXT   'REPLACES'
FTAB    18
FTEXT   'TEXT'
FMEND
```

Result on output data set:

```
ALPHANUMERIC    STRING
REPLACES          TEXT
          ↑       ↑
          3      16
```

FEXIT *Format exit* **FEXIT**

Syntax: [identifier] ␣ FEXIT.

Type: Format list declaration.

Description: Editing is terminated
In an Edit and Write (EDWRT) or DISPLAY instruction, the edited buffer will be written to the output device. No output request is performed. When the buffer is empty FEXIT has the same effect as reaching the FMEND.

FHIGH *Format High intensity* **FHIGH**

Syntax: [identifier] ␣ FHIGH

Type: Format-list-declaration.

Description: The characters following FHIGH will be displayed with normal intensity if it was before low intensity.
This declaration is only valid for the video display PTS 6344 and when the format list, in which the declaration FHIGH occurs, is invoked by the DISPLAY instruction.
FHIGH results in the control character X'1F' being edited into the buffer.

FILLR — *Fill repeat* — FILLR

Syntax: [identifier] ␣ FILLR ␣ { value expression / 'string-character' } { ,value expression / ,decimal-integer }

Type: Format list declaration.

Description: String-character is copied an integer number of times (maximum 63) into the edit buffer.

Example: FILLR ' ', 5

FINP *Format Input* **FINP**

Syntax: [identifier] ␣ FINP ␣ column [APPL=value]

Type: Format list declaration.

Description: This declaration defines a general input field on the screen. The input field starts at the position defined by column, which is a value expression.
The APPL option defines a control value which can be transferred from this field to the program, with the GETCTL instruction. Value may range from -32,768 to 32 767.
The FINP declaration, should be immediately followed by a FCOPY or FMEL declaration.

FKI *Format Keyboard input* **FKI**

Syntax: [identifier] ␣ FKI ␣ column [,APPL=value] [,SCHK=value] [,MINL=value] [,MAXL=value] [,DUPL=data-item identifier] [,NUM][ALPHA] [,REWRT] [,ME] [,NEOI] [,NCLR] [,CTAB] [,VERIF]

Type: Format list declaration

Description: This declaration defines an input field on the screen which is to receive input from a keyboard via the DYKI instruction. The start of the field is defined by column, which is a value expression. The declaration, with its options, must be followed by an FCOPY or FMEL declaration, which contains the input field belonging to data-item.

Options	Significance
	Data item must be of the type string or decimal.
MAXL=value	Value can be obtained with the GETCTL instruction and must be in the range 0 to 127 inclusive. Default value for this option is zero.
ME	This option indicates a compulsory input field. It controls the instructions GETFLD, TUP, TDOWN, TLDOWN, TLEFT, TRIGHT, TBWD, TFWD. (This ME option can be tested by the TSTCTL instruction, if requested).
MINL=value	Value can be obtained with the GETCTL instruction and must be in the range 0 to 63 inclusive. Default value for this option is zero.
ALPHA	This option controls the DYKI instruction and allows alpha numeric characters to be entered for this input field. Default is the NUM option. (This ALPHA option can be tested by the TSTCTL instruction, if requested).
APPL=value	Value can be obtained with the GETCTL instruction and must be in the range -32 768 to 32767. Default value for this option is zero. *
CTAB	Cursor setting is prohibited. This option controls the tabulation instruction TUP, TDOWN, TLDOWN, TLEFT, TRIGHT, TBWD, TFWD and THOME. (This CTAB option can be tested by the TSTCTL instruction, if requested).
DUPL=data-item-identifier	The contents of the data-item referenced by data-item-identifier can be obtained with the DUPL instruction.
NCLR	This option controls the ERASE instruction and prevents erasing of input fields. (This NCLR option can be tested by the TSTCTL instruction, if requested).

FKI *continued* FKI

Options	Significance
NEOI	The maximum number of input characters (MAXL) is accepted without a termination key. This option controls the DYKI instruction. (This NEOI option can be tested by the TSTCTL instruction, if requested).
NUM	This option controls the DYKI instruction and allows numeric characters to be entered for this input field. Either NUM or ALPHA may be present as option. (This NUM option can be tested by the TSTCTL instruction, if requested).
REWRT	This option controls the UPDFLD instruction and allows redisplaying of the contents of the data item belonging to this input field. (This REWRT option can be tested by the TSTCTL instruction, if requested).
SCHK=value	Value can be obtained with the GETCTL instruction. Value is a value expression and ranges from 1 to 7 inclusive. Default value for this option is zero. *
VERIF	This option indicates that the input field is subject for verification. It does not control any format control instruction but is obtained by the TSTCTL instruction.

* Note that the values used in the APPL and SCHK options are defined by the user for use in the program outside of the format list. Their values and use are therefore completely application-dependent.

FLINK *Format link* **FLINK**

Syntax: [identifier] ␣ FLINK ␣ data-item-identifier / format-list-identifier

Type: Format list declaration

Description: A format list as indicated by format-list-identifier is called as a sub-format, and editing continues according to this subformat. Upon the end of the subformat, editing is resumed at the next format-list-declaration. *
Instead of a format-list-identifier, operand-1 may be a reference to a string data-item. This data-item must contain format-list characters as present in the format-literalpool. (output CREDIT linker). Item size must be great enough to contain these characters. The CALL FMOVE instruction may be used to fill the data-item.

Example:
```
FCOPY = C'NEW BALANCE:'
FLINK   FRM012
FTEXT   'TOTAL:'
```

* Note that the new format list will be edited into the output buffer at the current position, the first two characters (the control characters) being omitted. If the new format list is required to start on a new line, the FLINK declaration should be preceded by FEOR and FCOPY (with control characters as relevant). The same is true for the format-list-declaration following the FLINK; it will be edited into the output buffer at the current position.

FLOW *Format Low Intensity* **FLOW**

Syntax: [identifier] ␣ FLOW

Type: Format list declaration.

Description: The characters following FLOW will be displayed with low intensity. This declaration is only valid for the video display PTS 6344 and when the format list, in which the declaration FLOW occurs, is invoked by the DISPLAY instruction. FLOW results in the control character X'1E' to be edited into the buffer.

FMEL *Format element* **FMEL**

Syntax: [identifier] ␣ FMEL ␣ picture-string, data-item-identifier

Type : Format list declaration.

Description : The decimal data item indicated by operand-2 is edited according to picture-string.
Editing is done from right to left.

Picture-string characters:

Character	*Significance*	*Example*
A	Skip if space (left-adjust to leading digit)	Picture 'AAA999' Data item BFF0456 RESULT 0456
B	Insert a blank space	Picture '99B99' Data item B06521 RESULT 65 ␣ 21
E	Enter the character following E into the data item. Any ISO-7 character may be entered, except a single quote or backslash	Picture '99E–99' Data item BO1912 RESULT 19–12
F	Insert character following F, before the next printed digit but after suppression of leading zeros and spaces. Any ISO 7 character may be entered, except a single quote or backslash	Picture 'F*ZZZ9V99' Data item B001053 RESULT *10.53
P	Skip this position	Picture 'P99' Data item B543 RESULT 43
T	Skip if space or leading zero (left adjusted to leading non-zero digit)	Picture 'TTT999' Data item BFFF0456 RESULT 456
V	Insert decimal point (not roomless). A decimal point preceding the leftmost digit will be replaced by asterisks in all leading positions.	Picture '99V99' Data item B00123 RESULT 01.23 Picture '**V9' Data item DF01 RESULT ***1
X	Print alphanumeric (space is printed as space and digits as digits)	Picture 'XXX' Data item BF12 RESULT ␣ 12

FMEL *continued* FMEL

Character	*Significance*	*Example*
Y	Enter alphanumeric if data item is non empty, else enter a space.	
Z	Leading zeroes are replaced by spaces	Picture 'ZZZ99' Data item B00123 RESULT ␣␣123
0	Insert zero	Picture '9909' Data item B123 RESULT 1203
9	Print digit (see X)	Picture '999' Data item BF12 RESULT 012
+	Print a + or − sign [2]	Picture '999+' Data item D123 RESULT 123+ Picture 'F+ZZZ' Data item DF01 RESULT ␣␣−1 Picture 'F+**V9' Data item DF11 RESULT **1.1
−	Print a − sign if the [2] data item is negative. Otherwise print space.	Picture '999−' Data item B123 RESULT 123␣
*	Replace leading zero or space by asterisk	Picture '***99' Data item B00123 RESULT **123
.	Insert roomless point [1]	
,	Insert comma [1]	Picture ''99,99' Data item B01234 RESULT 12,34

(1) If leading zeroes or spaces are being suppressed, any comma or decimal point (roomless or normal) occurring before the first non-suppressed digit will also be suppressed.

(2) + and − may be declared as floating, the function is further the same.

FMELI *Format element immediate* **FMELI**

Syntax: [identifier] ␣ FMELI ␣ picture-string, data-item-identifier

Type: Format list declaration.

Description: The decimal data item referenced by data-item-identifier is to be edited according to picture string.

The picture-string is included in the format pool and not in the picture pool. This is the only difference with the FMEL declaration.

For picture details, see FMEL.

FMEND *Format end* **FMEND**

Syntax: ␣FMEND␣

Type: Format list declaration.

Description: This declaration indicates the end of a format list.

FNL *Format next line* **FNL**

Syntax: [identifier] ␣ FNL

Type: Format list declaration.

Description: When this format list declaration occurs in a format list used by the EDIT instruction, editing will be terminated.
EDWRT and DISPLAY instructions using a format list in which a FNL declaration is present, will result in the following actions:

1. an output request is done for the current contents of the buffer, except when the buffer is empty.
2. A space character is inserted in the first position of the buffer and one line spacing control character space, is inserted in the second position. (One line feed). However, the logicical tabulation position is counted as one.
3. The control characters "low intensity" (X'1E'), is edited in the third position only, when the video display PTS 6344 and the instruction DISPLAY are used.

FNUL *Format No underlining* **FNUL**

Syntax: [identifier] ␣ FNUL

Type: Format-list-declaration.

Description: The characters following FNUL will be displayed with no underlining, if it was before underlining. This declaration is only valid for the video display PTS 6344 and when the format list, in which the declaration FNUL occurs, is invoked by the DISPLAY instruction.
FNUL results in the control character X'13' being edited into the buffer.

FRMT *Format* **FRMT**

Syntax: format-list-identifier ␣FRMT␣

Type: Format list declaration.

Description: This declaration indicates the beginning of a format list.

FSL *Format start line* **FSL**

Syntax: [identifier] ␣ FSL

Type: Format list declaration.

Description: When this format list declaration occurs in a format list used by the EDIT instruction, editing will be terminated.
EDWRT and DISPLAY instructions using a format list in which a FSL declaration is present, will result in the following actions:

1. An output request is made for the current contents of the buffer, except when the buffer is empty.
2. A space character is inserted in the first position of the buffer and one line spacing control character '+', is inserted in the second position. However, the logical tabulation position is counted as one.
3. The control character "low intensity" (X'1E) is edited in the third position only, when the video display PTS 6344 and the instruction DISPLAY are used.

FTAB *Format tabulation* **FTAB**

Syntax: [identifier] ␣ FTAB ␣ value-expression

Type: Format list declaration

Description: The pointer for the buffer is set to the position specified by the value-expression. This column may be to the right or to the left of the current position. The positions in the buffer between the current pointer and the new pointer are filled with space characters. Editing proceeds from the new pointer.
The first position in the buffer is counted as one when setting the pointer.
When calculating the tabulation position, the format-list-declarations FSL, FNL, FHIGH, FLOW, FUL, FNUL, FINP, and FKI each occupy one character in the buffer.

Example:

```
SIXT    FTAB    16
ONE     EQU     1
FIVE    FTAB    ONE+4
```

FTABLE *Format table generation.* **FTABLE**

Syntax: [format-table-identifier] FTABLE format-list-identifier, [,format-list-identifier] ...

Type: Format table declaration

Description: A one dimensional array of format-list-identifiers is declared, which may be referenced in instructions expecting a format-list-reference. Format lists referenced in the format table declaration may not be passed as a parameter. In EDSF and PFRFT instructions only the name of the complete format table can be passed.
In the heading of a subroutine the formal parameter name for the format table must be followed by two brackets.
Following the PROC directive, the formal parameter name must be mentioned in a PERMT directive, even if no two byte addressing is selected. The rules which apply to elements in a one dimensional array are also valid for format tables.

Example:

```
FTB1    FTABLE    FORM1,FORM2,FORM3

EDWRT   DSVDU,FTB1(INDEX)

PROC    $FFTAB ( )
PERMT   $FFTAB
```

FTEXT *Format immediate text* **FTEXT**

Syntex: [identifier] ␣ FTEXT { 'string-character ... ' / X' hexadecimal-digit ...' }

Type: Formal list declaration

Description: String-character(s) or hexadecimal-digits are copied into the output buffer. Copying is done on character base. Not allowed characters are the quote or backslash.

Example:

```
FTEXT    'NEW BALANCE'
FTEXT    X'4E554553'
FTEXT    X'30
```

FUL *Format underlining* **FUL**

Syntax: [identifier] ␣ FUL

Type: Format list declaration.

Description: The characters following FUL will be displayed with underlining, if it was before no underlining. This declaration is only valid for the video display PTS 6344 and when the format-list, in which the declaration FUL occurs, is invoked by the DISPLAY instruction.
FUL results in the control character X'12' being edited into the buffer.

KTAB *Key table* **KTAB**

Syntax: Key-table-identifier ␣KTAB␣ key-value [,key-value] ....

Type: Key table declaration.

Description: A key value in the key value list is used as a terminating character (end of record key) in the keyboard input instructions.
If a value expression is used in key value list, only value type X may be used.

Example:
```
KTAB1   KTAB   KCORR,KMUL,KDIV
KTAB2   KTAB   X'0D',X '30' + 2
```

PLIST *Parameter list* **PLIST**

Syntax: ␣ PLIST ␣ actual-parameter [,actual-parameter] . . .

Type: Parameter declaration.

Description: This declaration specifies parameter(s), which have to be passed to a subroutine. The PLIST declaration follows immediately a PERFI instruction.
A literal constant of the type 'X' is not allowed as parameter.

2 PROGRAM TESTING

2.1 Introduction

A CREDIT program may be input to the FTS system via cards, cassette, flexible disk or console typewriter. After input the source module is held on disk. All the processors and utilities described in Part 2 of this manual read input from disk and write output to disk.

The diagram on page 2.1.2 illustrates the sequence of processes needed to develop and run an executable program from CREDIT source modules.

Each source module is processed separately by the CREDIT Translator. The Translator produces intermediate object code modules. The instructions in these modules use a byte oriented addressing system. Each module may contain references to:

- Labels in the same module
- Literals, formats, key tables and pictures in the same module and/or in the same segment.
- Labels in other CREDIT modules, in the same segment and/or in other segments.
- Assembler application modules.
- Assembler system routines.

The first three types are satisfied by the CREDIT linker.

The remaining types of reference are satisfied by the Linkage Editor. This processor builds an application load module from the following object modules:

- Object modules from the CREDIT linker.
- Assembler application modules (if referenced).
- Assembler system routines.
- CREDIT interpreter.
- CREDIT debugging program (if requested).

CREDIT code cannot be executed directly. Each instruction must be interpreted by the CREDIT Interpreter. The routines within the Interpreter actually perform the functions specified in the CREDIT code. For this reason the Interpreter is built into the load module by the Linkage Editor.

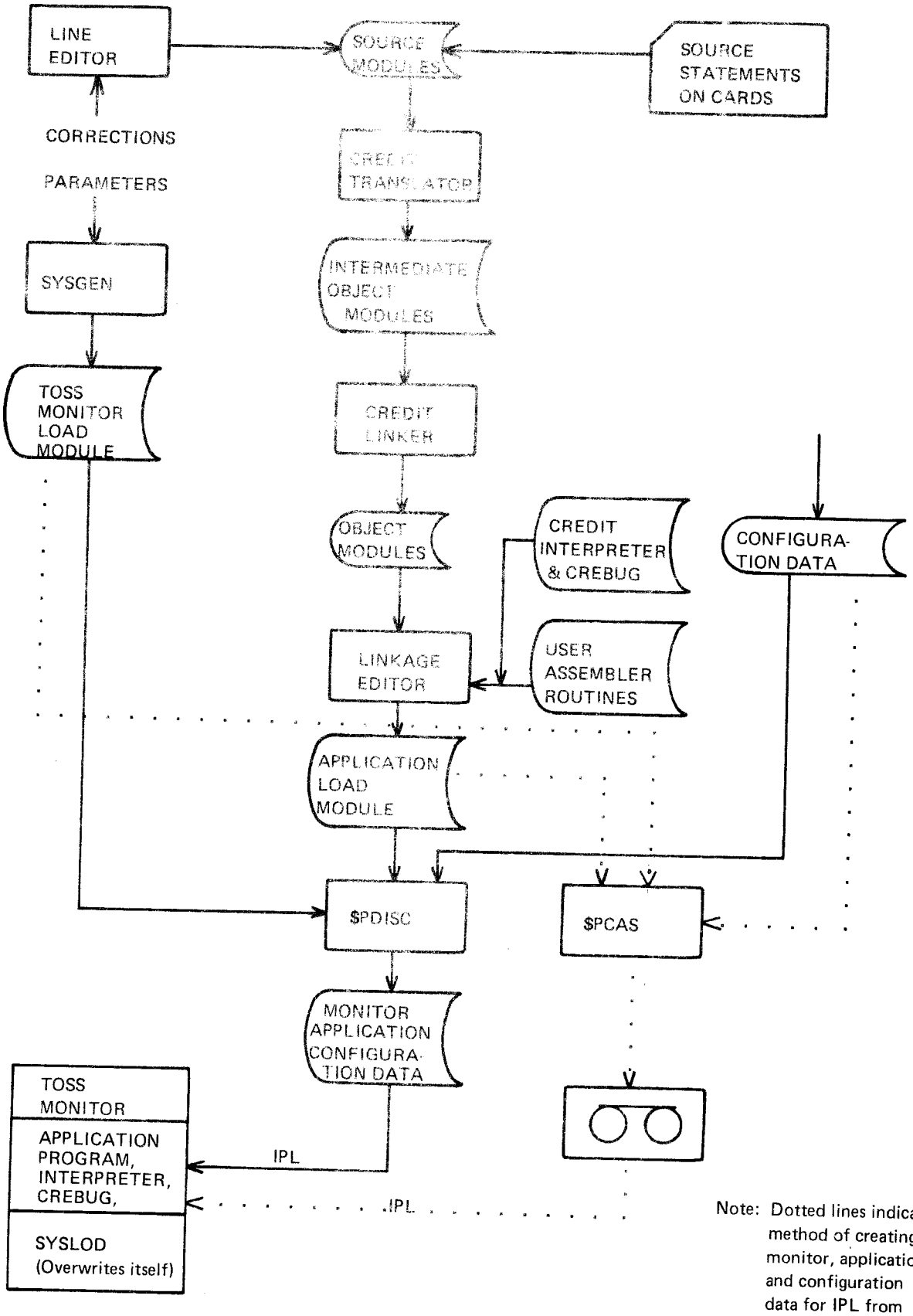
The CREDIT Debugging Program, if required, must also be built into the load module. The Debugging Program is an interactive diagnostic task which, if present, is executed in parallel with the CREDIT program being tested. Via the Debugging Program the programmer can monitor and control the execution of his program.

When the load module is loaded into memory for execution the work blocks, stacks, data set buffers and task control areas required for a particular system, are set up. This is done by the System Loading Program (SYSLOD).

The CREDIT translator, CREDIT linker and linkage editor are all run under the DOS 6800 monitor. The load module produced by the linkage editor, however, must be run under the FTS monitor.

When control is passed to the load module application load module, control is handed to the System Loading Program (SYSLOD). When configuration is complete, control is handed to the application program. Testing may then be carried out using one or more work positions. If the CREDIT debugging program is being used, execution can be controlled from the work position.

Development of a CREDIT Application Program



Note: Dotted lines indicate method of creating monitor, application and configuration data for IPL from cassette.

If any application program errors are detected during testing, one or more source modules will have to be corrected. This may be done via the Line Editor – an interactive text editor. Each corrected source module must then be re-processed by the CREDIT Translator. The whole program must then be processed by the CREDIT Linker and Linkage Editor prior to re-testing.

TOSS system software comprises the following components:

– Monitor
– System Loading Program
– CREDIT interpreter
– [CREDIT debugging program]
– [Assembler debugging program]

These software components are not described in a separate manual. Information concerning TOSS System Software which is needed by CREDIT programmers is contained in this manual.

The following software components, though part of DOS6800 System Software, are discussed in this manual:

- CREDIT Translator
- CREDIT Linker

They are discussed in this manual because they are used by CREDIT programmers only. The remaining DOS 6800System Software components used by CREDIT programmers, notably the Linkage Editor and Line Editor, are described in the DOS6800 System Software PRM (M11).

2.2 CREDIT Translator

2.2.1 *Introduction*

The CREDIT Translator is a processor which converts CREDIT source statements into intermediate object code. Source modules are translated separately, resulting in the production of individual object modules. References between object modules and references to external routines etc., are not resolved by the Translator.

Readers of Section 2.2 should be familiar with the following DOS6800 System Software concepts:

- Control Command
- Processor
- EOF mark
- Source input device
- Temporary source file
- Temporary object file

These concepts are explained in the DOS6800 System Software PRM (M11).

2.2.2 *Running the Translator*

Source modules must be read into the System by issuing the control command RDS (read source). RDS will read the source module from the input device (card reader, cassette or console keyboard) and will create a temporary source file. The module must be terminated by an :EOF mark. If the module has been read into the System previously and kept (control command KPF), a RDS command will not be necessary.

It is strongly recommended that all temporary object files are scratched (and kept if necessary) before the Translator is executed. This will ensure that the output object modules will not be corrupted by existing files.

The translator is called into execution by the following control command:

TRA␣ { /S / name } [,NL]

where: /S indicates that the input source module is in the temporary source file.
name is the name of a source file in the library of the current user identifier. It indicates that the input source module will be found in that file.
NL indicates that no listing of the module is required. Error messages are always printed.

The intermediate object module created by the Translator is written into the temporary object file. If this file already contains object modules, the following action is taken. If it has not been closed by an EOF mark, the intermediate object module is written *after* the information already held in the file. If it has been closed by an EOF mark, a new temporary object file is created and the old one is deleted.

2.2.3 *Translator Listing*

2.2.3.1 *General*

During translation the Translator generates a listing in three parts. Part one contains the CREDIT source statements, intermediate object code and error messages. Parts two and three contain the data item name table and the procedure label table. The following sections describe these parts.

The listing can be suppressed if the NL option is specified on the TRA control command. In this case, only the error messages will be printed.

2.2.3.2 *CREDIT code and Error Messages*

The format of this part is shown in the following example. The example is taken from the procedure division. The data division listing is slightly different. The differences are noted in the explanation which follows.

At the left of the listing under the heading LOC is the location counter. This is a four digit hexadecimal counter which is stepped by one each time a byte of intermediate object code is generated. In the data division a two digit hexadecimal counter called IX (for index) is used.

The next eight items, under the headings OC (operation code) and OPERANDS, comprise the generated interpretive instructions. Each item is a two digit hexadecimal code. The significance of these codes is described for each instruction in the Instruction Reference Section (1.4.8). Object code is not listed in the data division.

The item under the heading LINE is a four digit decimal line counter.

The remaining items are self-explanatory. They comprise CREDIT source statements.

Errors in the source module are reported by the Translator. One of the following messages is printed immediately after the line containing the error:

01	Memory overflow (job aborted)
02	Sequence error
03	Directive missing
04	Syntax error
05	Length truncation (no error accumulation)
06	Multidefined
07	Undefined
08	Unexpected value
09	Undefined type
0A	Unexpected type
0B	Illegal constant
0C	Lit pool overflow
0D	Label missing
0E	Illegal value def
0F	Illegal const length
10	Illegal const type
11	Too many blocks
12	Too many data items.
13	Block size overflow
14	Too many datasets
15	Too many parameters
16	Too many start stmts
17	Illegal dimension.
18	Too many values.
19	Out of range.
1A	Unspecified parameter.
1B	Parameterlist overflow.

In addition to the error message an asterisk is printed to show the position at which the error occurred.

An error count is maintained by the Translator and is printed after the END directive.

If a fatal error occurs (I/O error, table overflow, etc), the source input is read until an EOF mark is encountered and the following message is printed:

FATAL ERROR HAS OCCURRED. NO OBJECT CODE PRODUCED.

The object file is then deleted.

2.2.3.3 *Data Item Name Table*

The data item name table is listed immediately after the CREDIT code and error messages. For each data item declared in a work block it contains the following:

NAME This is the data item identifier.

REF This is the index number assigned by the Translator. Index numbers are printed to the left of the data item declarations, under the heading 'IX'.

TYPE This is the data item type specified in the data item declaration. The following mnemonics are printed under the TYPE heading:
BCD (decimal), BIN (binary), BOL (boolean) and STR (string).
A letter U following one of these mnemonics indicates that the data item is not referenced in this module.

2.2.3.4 *Procedure Label Table*

The procedure label table is printed immediately after the data item name table. For each identifier appearing in the procedure division it contains the following:

NAME This is an identifier specified by the programmer in the procedure division, or it is the name of a System routine referred to by the generated object code.

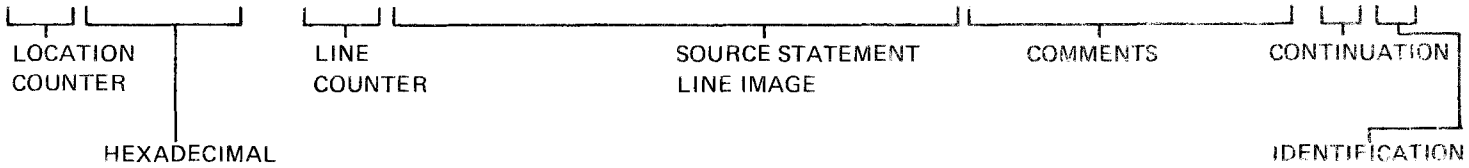
REF This may be an index to a format list, a key table or an external table. It may be a value specified in an equate directive. It may be the contents of the location counter (if the identifier belongs to an instruction or PROC directive).

TYPE This indicates the type of identifier. The following mnemonics are used:

ADR	Address of an instruction
EQU	Equate directive
EXT	External label
FOR	Format list
KEY	Key table
PRO	PROC directive
FLB	Format label (address of a format item)
FTB	Format table

A letter U following one of these mnemonics indicates that the identifier is not referred to in this module.

```
CREDIT TRANSLATOR  REL 3.1 780520 *  PROCEDURE DIVISION          IDENT  OPCLOS * 060477 * PAGE 0006

LOC  OC OPERANDS    LINE LABEL       OPCOD OPERANDS              COMMENT                C      1D
                      0336             EJECT
                      0337 *
                      0338 *
                      0339 *           ENTRY TERMINAL CLOSE MODULE
                      0340 *
                      0341 *
                      0342  OPCL05
0000   30 XX 85 FF    0343             EDWRT DSTPJT,FRM003          PRINT ASTERIX
0004   30 XX 85 FF    0344             EDWRT DSTPJT,FRM014          PRINT CLOSE
                      0345
                      0346  OPC010
0008  XX KK LL LL     0347             PERF  READN,KTAB3,=W'4',=W'4' READ OPERID
000C  5F 06           0348             SB    OPC010
                      0349
000E   20 60 42 05    0350             CB    EQ,CASHID,INPUT,OPC100 CORRECT IDENTITY
0012  XX 12 LL        0351             PERF  ERROR,TLAMP3,KTAB4
0015  5F 0F           0352             SB    OPC010
```



CREDIT TRANSLATOR LISTING EXAMPLE

2.3 CREDIT Memory Management Linker

2.3.1 *Introduction*

The CREDIT memory management linker is a three pass processor which converts intermediate object code, produced by the CREDIT translator into object code which can be processed by the Linkage Editor. The CREDIT linker is capable of linking both unsegmented and segmented programs.

Intermediate object modules may contain references to:

- Labels in the same module.
- Literal constants, formats, key tables, pictures in the same module.
- Labels in other CREDIT modules in the same segment.
- Labels in other CREDIT modules in other segments.
- Assembler application modules.
- Assembler System routines.

The first three types of reference, when present in the same segment, are satisfied by the CREDIT linker.
The remaining types of reference must be satisfied by the Linkage Editor.
To build up the segments, different possibilities exist which are the same when using extended main memory, secondary memory or a combination of both.

Readers should be familiar with the following DOS6800 System Software concepts:

- Control command
- User library
- User identifier
- Temporary object file

These concepts are explained in the DOS6800 System Software PRM (M11).

2.3.2 *Building up segments*

After translation of the different CREDIT modules a number of intermediate object modules have been created. All these modules together building up a CREDIT application, are input to the CREDIT linker (TLK).
The CREDIT linker has to know which modules should be contained in the segments. This is specified by the user by means of the ordering of the modules as input to the linker. In the TLK command is a parameter (n or mK) defining the maximum segment size to be used.
However, the NOD command (node) can be used to force an immediate end of a segment and also to define the segment as main memory resident (NOD ␣ R) or belonging to the common area, segment 00 (NOD ␣ C). The common area, segment 00, is always present in main memory and will contain the data division, the interpreter, assembler sub-routines and / or user routines. The size of the common area is variable and not dependent on the size parameter in the TLK command. Segment 00 is automatically created.
When the NOD command is used without specifying R or C as parameter, the segment will be disk resident. When the NOD command is not used, the segments will be disk resident.

The output from the CREDIT linker is placed in temporary object file (/0) and divided into different modules as: data division common part and segments. The user can extend the common part, segment 00, with the NOD L,C command. The modules are grouped into segments in the order in which they are included with the INC command. The segments are numbered from 1 upward.

Some examples showing the use of NOD and TLK for different systems.

a. System with 64K Byte main memory.

```
INC   MOD1
INC   MOD2
INC   MOD3
TLK   U,M,X
```

b. System with 64K Byte main memory and use of secondary memory. (Disk, flexible disk).

```
                                        Size
INC   MOD1                              (3.5Kbytes)
INC   MOD2,   USER1                     (0.7Kbytes)
INC   MOD3,   USER2                     (0.5Kbytes)
INC   MOD4                              (2Kbytes)
INC   MOD5                              (2Kbytes)
INC   MOD6,   USER3                     (1Kbyte)
INC   MOD7                              (2Kbytes)
TLK   U,M,4K
```

Four segments are created by the linker, and all are disc resident.
(Segment zero always resides in main memory.)

Segment 1	Contains	MOD1	(3.5K)
Segment 2	Contains	MOD2, MOD3, MOD4	(3.2K)
Segment 3	Contains	MOD5, MOD6	(3K)
Segment 4	Contains	MOD7	(2K)

By means of altering the sequence of the INC commands, the user can optimize his program segments. In this example only 50% of sement 4 is filled.
When e.g. MOD5 must be present in segment 0, the following sequence of commands has to be specified:

```
                                        Size
NOD   C
INC   MOD5                              (2Kbytes)
NOD
INC   MOD1                              (3.5Kbytes)
INC   MOD2,   USER1                     (0.7Kbytes)
INC   MOD3,   USER2                     (0.5Kbytes)
INC   MOD4                              (2Kbytes)
INC   MOD6,   USER3                     (1Kbyte)
INC   MOD7                              (2Kbytes)
TLK   U,M,4K
```

Segment 1 contains: MOD1
Segment 2 contains: MOD2, MOD3, MOD4
Segment 3 contains: MOD6, MOD7,.

MOD5 is now included in the common area, segment zero.

When also MOD1 must be main memory resident, but not in segment 0, then the following command sequence can be used:

```
                                    Size
NOD     C                           (2Kbytes)
INC     MOD5                        (2Kbytes
NOD     R
INC     MOD1                        (3.5Kbytes)
NOD
INC     MOD2,   USER1               (0.7Kbytes)
INC     MOD3,   USER2               (0.5Kbytes)
INC     MOD4                        (2Kbytes)
INC     MOD6,   USER3               (1Kbyte)
INC     MOD7                        (2Kbytes)
TLK     U,M,4K
```

Segment 1 contains: MOD1 (3.5Kbytes)
Segment 2 contains: MOD2,MOD3,MOD4 (3.2Kbytes)
Segment 3 contains: MOD6, MOD7 (3Kbytes)

c. System with extended main memory, up to 256Kbytes.

```
                                    Size
INC     MOD1                        (3.5Kbytes)
INC     MOD2,   USER1               (0.7Kbytes)
INC     MOD3,   USER2               (0.5Kbytes)
INC     MOD4                        (2Kbytes)
INC     MOD5                        (2Kbytes)
INC     MOD6,   USER3               (1Kbyte)
INC     MOD7                        (2Kbytes)
TLK     U,M,4K
```

Four segments are created by the linker, and are assumed to be disk resident. Because an extended main memory is used, all segments will be main memory resident. The composition of the segments is as mentioned in example b.
The system loader SYSLOD will discover the difference when a system with extended main memory, secondary memory or a combination of both is used.

d. Systems with extended main memory (up to 256 bytes) and secondary memory.

Examples b) and c) may be combined.

2.3.3 *Running linker*

The CREDIT linker reads intermediate object modules from temporary object file and from the library of the current user identifier. The syntax of the TLK command is:

TLK ␣ [N|S|U][,X] [,M] [,n[,mK]

N The system or user /ØBJCT files do not need to be scanned.

U Only the user /ØBJCT files will be scanned.
S Only the system /ØBJCT file has to be scanned.

Default value: Both /ØBJCT files will be scanned.
The user /ØBJCT file will be scanned first, then the system /ØBJCT file and then the user /ØBJCT file again.

X Indicates that a cross reference listing is required.

Default value: No cross reference will be printed.

M The listing of the map, which consists of a listing of the module names and the relative start addresses, address pools and statistics per segment.

Default value: No map will be printed.

n The required segment size in bytes.
mK The required segment size in K bytes. (m x 1024 bytes)

Default value: The program will be unsegmented.

2.3.3.1 *CREDIT modules in the system library*

When CREDIT modules have to be linked from the System library (USERID:SYSTEM), first a Generate Object Directory command (GOD) must be executed for this library before the TLK command can be executed. (In any other user than SYSTEM.)

The following listings are produced by the CREDIT memory management linker per segment:

- Segment 0 (Common part)
  load map
  long branch table
  call table
  perform table
  literal pool
  key table pool
  picture pool
  format pool
  } can be excluded by not using 'M' in the TLK command.
  Linker statics for this segment
- Segment n
  loadmap
  long branch table
  perform table
  literal pool
  picture pool
  format pool
  } can be excluded by not using 'M' in the TLK command.
  Linker statics for this segment
- Total
  segment map
  cross reference
  } can be excluded by not using 'X' in the TLK command.
  Linker statics for the whole program.

The load map includes a list of error reports. Error reports will be listed even if the load map listing has not been requested.

2.3.3.2 *Load Map*

The load map indicates the displacement of each module within a segment. It also contains the linker (TLK) error reports. The format of the load map is shown in the following example:

```
-------------------------------------------------------------
* CREDIT CODE LINKER PRR 4.1 790410 * LOAD MAP  SEGMENT  02
-------------------------------------------------------------

LOC       MODULE    ERROR          COMMENT

000E      MOD3                     TRA 4.1  99-99-99 F1 01111
006D      MODUL4                   TRA 4.1  99-99-99 F1 01111
0090      MODUL6                   TRA 4.1  99-99-99 F1 01111
00B9      MODUL7                   TRA 4.1  99-99-99 F1 01111
00C8      MODUL8                   TRA 4.1  99-99-99 F1 01111
```

where: LOC is the displacement of the module within the segment.
MODULE is the module name.
ERROR is the error number followed by a type, number and clear text.

Error type may be:

E – User Error
I – Internal error or input inconsistency
W – Warning, no updating of error counter.

The following error reports may be printed:

ERROR NUMBER	ERROR TYPE	Additional Information	Text (Significance)
0	I		END OF MEMORY No more work space available
1	E		SYMBOL TYPE CONFLICT LB, CALL or PERF mixed up
2	I	XXXX	ILLEGAL INPUT XXXX is a hexadecimal presentation of 1st and 3rd character in cluster. Input from translator not expected.
3	I	XXXX	LOAD ADR INCORRECT XXXX is a hexadecimal presentation of load address from the cluster.
4	W	DDDD	UNREFERENCED LITERAL DDDD is a decimal presentation of the number of unreferenced literals.

ERROR NUMBER	ERROR TYPE	Additional Information	Text (Significance)
5	I	DDDD	UNDEFINED LITERAL DDDD is a decimal presentation of the number of undefined literals.
6	E		NO START ADDR Start address declaration not found in the data division.
7	E		DBL DEF MODULES Double defined modules
8	E	DDDD	UNSATISFIED EXTERNAL DDDD is a decimal presentation of the number of unsatisfied externals. The unsatisfied externals are printed on the load map.
9	I	XXXX	MODULE LENGTH ERROR XXXX is a hexadecimal presentation of the difference between requested and available workspace.
10	E	DDDD	TRANSLATION ERROR DDDD is a decimal presentation of the number of translating errors.
11	E	XXXX	WRONG TRANSLATOR RELEASE XXXX is a hexadecimal representation of the lowest acceptable level of the CREDIT translator, in the form RRLL RR = Release number LL = Level number
12	E	C	ADDRESS TABLE OVERFLOW C is a character representing the address type L (Long Branch), C (Call) or P (Perform).
13	E	C	LITERAL DISPLACEMENT OVERFLOW C is a character representing the literal type L (Literal), K (key table), P (picture), or F (format).
14	E	C	TOO MANY LITERALS C is a character representing the literal type L (literal), K (key table), P (picture), or F (format).
15	E		FORMAT LENGTH ERROR
16	E		MULTI DEF ENTRY Entry name defined in more than one module.
17	E	C	NOD TYPE ERROR C is a character representing the NOD type. NOD type not C, D or R.

ERROR NUMBER	ERROR TYPE	Additional Information	Text (Significance)
18	E	XXXX	MODULE LONGER THAN SEGMENT SIZE XXXX is a hexadecimal representation of the module length. Increase segment size.
19	I		IDENT MISSING
20	E		ADDRESSING MODE CONFLICT One byte or two bytes addressing mode of literals mixed up.
21	E		NOD SEQUENCE ERROR The NON record "NOD C" does not appear in the beginning of the object input.

2.3.3.3 *Call table*

This table contains all references to external routines (CALL instruction) which could not be 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. The format of the call table is shown in the following example:

```
--------------------------------------------------------------
* CREDIT CODE LINKER PRR 4.1 790410 * CALL TABLE  SEGMENT  00
--------------------------------------------------------------

LOC   DATA      IX  SYMBOL  DEFINED

0002  ****      01  T:ASSI
0004  ****      02  T:KI
0006  ****      03  T:EDWR
0008  ****      04  T:DSC1
000A  ****      05  T:NKI
000C  ****      06  T:RREA
000E  ****      07  T:RWRI
```

where: LOC is the displacement of each table entry within segment zero.
DATA is the called address relative to the start of segment zero. It is generated by the linkage editor and is therefore not specified in the listing.
IX is the index value (maximum index is X'FF')
SYMBOL is the name of the external routine.
DEFINED is printed in this table.

2.3.3.4 *Long branch table*

In order to reduce the amount of memory required for a long branch instruction, linker (TLK) generates a table of destination addresses. Each time a long branch is encountered in the intermediate code, the linker places the destination address (i.e. segment number 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. The format of the long branch table is shown in the following example:

```
-------------------------------------------------------------
* CREDIT CODE LINKER PRR 4.1 790410 * LB TABLE  SEGMENT  01
-------------------------------------------------------------

LOC   DATA       IX  SYMBOL  DEFINED

0170  01 007A    01           MODULE
0174  01 0054    02           MODULE
0178  01 0118    03           MODULE
017C  01 009D    04           MODULE
0180  01 00F8    05           MODULE
0184  01 00E3    06           MODULE
0188  01 00C0    07           MODULE
018C  01 010F    08           MODULE
```

where: LOC is the displacement of each table entry within the segment.
DATA is the destination addres of the long branch. The first two digits specify the segment number and the next four specify the displacement within this segment. The difference between the four digit hexadecimal value, and the relevant module start address shown in the load map, gives the address of the destination within that module.

IX	is the index value used in the long branch instructions. It starts at one. (Maximum index is X'FF').
SYMBOL	is the statement identifier of the first instruction (location counter = 0) in the module containing the destination of the branch.
DEFINED	is the name of the module containing the destination.

2.3.3.5 *Perform table*

This table contains the address of each CREDIT subroutine which is called (PERF instruction) 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. The format of the perform table is shown in the following example:

```
* CREDIT CODE LINKER PRR 4.1 790410 * PERFORM TABLE   SEGMENT  01

LOC   DATA       IX  SYMBOL  DEFINED

0192  01 0037   01  VDU1    MOD1
0196  XX XXXX   02  KB1
019A  XX XXXX   03  VDU2
019E  01 0052   04  KB2     MODULE
01A2  XX XXXX   05  DISC
01A6  01 0030   06  STP1    MOD2
01AA  XX XXXX   07  VDU3
01AE  XX XXXX   08  VDU4
01B2  01 0046   09  STP2    MOD2
```

where: LOC is the displacement of each table entry within the segment.
DATA is the destination address of the perform. The first two digits specify the segment number and the next four specify the displacement within this segment. The difference between the four digit hexadecimal value, and the relevant module start address shown in the loadmap, gives the address of the destination within that module.

IX	is the index value. It starts at one (maximum index is X'FF').
SYMBOL	is the name of the subroutine. It only appears when the subroutine is not in the same module as the perform instruction.
DEFINED	is the name of the module which contains the subroutine.

2.3.3.6 *Literal pool*

The literal pool contains all the literals used in this segment. Each time a literal is encountered in the intermediate code it is replaced by an "index value" which points to the literal in the literal pool. The format of the literal pool is shown in the following example:

```
* CREDIT CODE LINKER PRR 4.1 790410 * LITERAL POOL  SEGMENT  02

IX     TYPE     LOC    DATA

10     BIN      00D8   0000
11     BIN      00DA   0004
12     BIN      00DC   0006
13     BIN      00DE   0008
14     BIN      00E0   0009
15     BIN      00E2   0016
16     BIN      00E4   0033
17     BIN      00E6   0040
18     BIN      00E8   0042
19     BIN      00EA   0200
1A     BIN      00EC   0410
1B     BIN      00EE   1410
1C     STR      00F0   07
1D     STR      00F1   202B
1E     STR      00F3   2030
1F     STR      00F5   203107
20     STR      00F8   41414115
21     STR      00FC   41414215
22     STR      0100   41414315
23     STR      0104   41414415
24     STR      0108   41414515
```

where: IX — is the index value. It starts at 10 or 4100 (maximum index is X'FF' or X'4FFF').

TYPE — indicates the value type of the literal. The following mnemonics are used:
BIN for value types X and W.
BCD for value type D.
STR for value type C.

LOC — is the displacement of each literal within the segment.

DATA — is the hexadecimal representation of the literal.

2.3.3.7 *Picture pool*

The picture pool contains all picture strings used in this segment. Each time a reference 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. The format of the picture pool is shown in the following example:

```
-------------------------------------------------------------
* CREDIT CODE LINKER PRR 4.1 790410 * PICTURE POOL  SEGMENT  01
-------------------------------------------------------------

IX     TYPE     LOC    DATA

10     PIC      01D1   393939
11     PIC      01D4   5A5A5A5A5A5A5A392C3939
12     PIC      01DF   3939452D3939452D39393939
13     PIC      01EB   5A5A565A5A5A565A5A392C39392B
```

where: IX — is the index value. It starts at 10 or 5100 (maximum index is X'FF' or X'5FFF').
TYPE — indicates the entry is a picture string (PIC).
LOC — is the displacement of each picture string within the segment.
DATA — is the hexadecimal representation of the picture string.

2.3.3.8 *Keytable pool*

The keytable pool contains all keytables used in the application program and is located in segment zero. Each time a reference to a keytable is encountered in the intermediate code, it is replaced by an "index value" which points to the keytable in the pool. The format of the keytable pool is shown in the following example:

```
-------------------------------------------------------------
* CREDIT CODE LINKER PRR 4.1 790410 * KEYTABLE POOL   SEGMENT  00
-------------------------------------------------------------

IX     TYPE     LOC    DATA

10     KEY      0010   031E1D19
```

where: IX — is the index value. It starts at 10 or 6100 (maximum index is X'FF' or X'7FFF').
TYPE — indicates the entry is a keytable. (KEY)
LOC — is the displacement of the keytable within segment zero.
DATA — is the hexadecimal representation of the keytable. First character in the keytable is the length indicator.

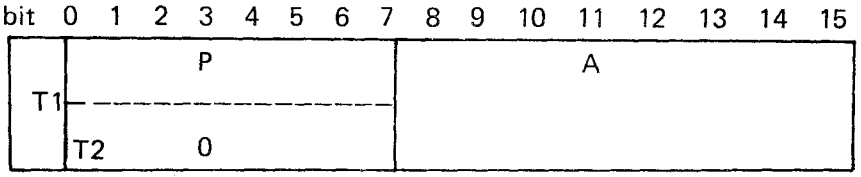
2.3.3.9 *Format pool*

The format pool contains all format lists used in the segment.
Each time a reference 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. The format of the format pool is shown in the following example:

```
* CREDIT CODE LINKER PRR 4.1 790410 * FORMAT POOL   SEGMENT  01                    * DATE 2857   * PAGE    9 *

IX    TYPE    LOC   DATA

10    FMT     01F9  C11BC027
11    FMT     01F0  C11BC026
12    FMT     0201  C11BC025
13    FMT     0205  C11B1322
14    FMT     0209  C11CC30F41555544B4F52495459 2E2E2E2E2E3A9E20C3054441544453A
15    FMT     0225  C11C9420C3192A2A5452414E5341435444 94F4E2043414E4345 4C4C45442A2AE8C11CA82AE8C11CA82A
16    FMT     024E  C11CA82AE8C11CC31F454E442044F4620444 1592C53455256494345204449534 34F4E5447 4E554544E8C11CA82A
                    E8C11CA82A
17    FMT     0280  C11CC30F5452414E5341435449 4F4E2E2E2E2E3A10219E20C3054441544453A1220E8C11CC11CC30E4F5045524154
                    4F5220434F44453ACD2A9E2DC30D4F4646494345204E4F3A313233E8C11CC3054E414D453ACD258620C3074144
                    524553533ACD26E8C11CC3054349545 93ACD27892DC30B4143434F554E5420 4E4F3ACD29E8C11CC3074140 4F55
                    4E543A1122902DC30C4E4557204241 4C414E43453A1123
```

where: IX — is the index value. It starts at 10 or 7100 (maximum index is X'FF' or X'7FFF').
TYPE — indicates that the entry is a format list (FMT) or format table FTB). The layout of FMT entries is explained below.
LOC — is the displacement of each format list in the pool within the segment.
DATA — is the hexadecimal representation of the format list or format table.

Each word in an FMT entry has the following layout:



Depending on the T1 and T2 bit, fields P and A or 0 and A have the following meaning:

T1 = 0; P field contains an index to a picture string (FMEL).
A field refers to decimal-data-item

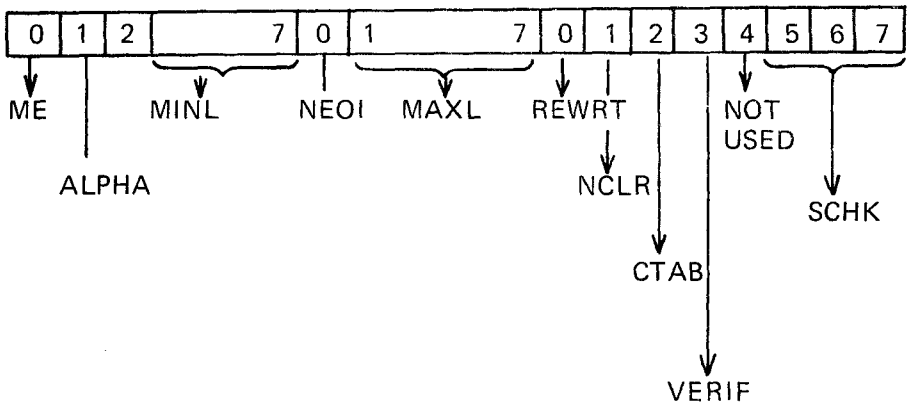
T1 = 1; 0 field contains a six-bit value, indicating how many times the character in the A-field has to be copied. (FILLR).

T1 = 1, T2 = 1;

Contents 0-field	Significance
00/01	A-field contains a reference to a string-data-item or literal (FCOPY).
03	A-field and following bytes contain ISO-7 characters (FTEXT).
04	A-field contains a tabulation value. (FTAB).
08	Character X'1F' edited into the buffer. A-field not used (FHIGH).
09	Character X'1E' edited into the buffer. A-field not used (FLOW).
0A	Character X'12' edited into the buffer. A-field not used (FUL).
0B	Character X'13' edited into the buffer. A-field not used (FNUL).
10	A-field contains a reference to a binary or decimal-data-item. The byte following the A-field contains a displacement. (FBZ).
11	A-field contains a reference to a binary or decimal-data-item. The byte following the A-field contains a displacement. (FBP).
12	A-field contains a reference to a binary or decimal-data-item. The byte following the A-field contains a displacement (FBN).
14	A-field contains a reference to a binary or decimal-data-item. The byte following the A-field contains a displacement (FBNZ).
15	A-field contains a reference to a binary or decimal-data-item. The byte following the A-field contains a displacement (FBNP).
16	A-field contains a reference to a binary or decimal-data-item. The byte following the A-field contains a displacement (FBNN)
18	A-field contains a displacement (FB).
1A	A-field contains a reference to a boolean-data-item. The byte following the A-field contains a displacement (FBF).
1B	A-field contains a reference to a boolean-data-item. The byte following the A-field contains a displacement (FBT).
1C	A-field contains a reference to a binary-data-item. (FCW).
1D	A-field contains a reference to a literal (FCW).
1F	A-field contains a reference to a subformat list (FLINK).
20	A-field not used (FSL).
21	A-field not used (FNL).
28	A-field not used (FEOR).
29	A-field not used (FEXIT).

2C	A-field contains the tabulation position (FINP).
2E	A-field contains the tabulation position, the following byte contains the right halfword of the application (APPL) control field (FINP).
2F	A-field contains the tabulation position, the following 2 bytes contain in sequence: a) left halfword of the application (APPL) control field b) right halfword of the application (APPL) control field (FINP).
30	A-field contains the tabulation position. The three following bytes constitute the standard input control field (FKI).

Layout standard control field (FKI).



32	A-field contains the tabulation position. The following byte contains the right halfword of the application (APPL) control field. The next three bytes are the standard control field bytes (see also 30) (FKI).
33	A-field contains the tabulation position. The following bytes contain in sequence: a) left halfword of the application (APPL) control field b) right halfword of the application (APPL) control field c), d) and e) are the standard control field bytes (see also 30) (FKI).
38	A-field contains the tabulation position. The following bytes contain in sequence : a) duplication data-item (DUPL) reference. b), c) and d) are the standard control field bytes (see also 30) (FKI).
3A	A-field contains the tabulation position. The following bytes contain in sequence: a) right halfword of the application (APPL) control field b) duplication data-item (DUPL) reference c), d) and e) are the standard control field bytes (see also 30) (FKI).

3B	A-field contains the tabulation position the following bytes contain in sequence: a) left halfword of the application (APPL) control field b) right halfword of the application (APPL) control field c) duplication data-item (DUPL) reference d), e) and f) are the standard control field bytes (see also 30) (FKI).

2.3.3.10 *Linker statistics per segment*

The format of the linker statistics listing per segment, is shown in the following example. The contents of the listing are self-explanatory.

```
* CREDIT CODE LINKER PRR 4.1 790410 * LINKER STATISTICS   SEGMENT  00

  ALL VALUES DECIMAL

                 LB TABLE:      0 BYTES.      0 ENTRIES
               CALL TABLE:     14 BYTES.      7 ENTRIES
            PERFORM TABLE:      0 BYTES.      0 ENTRIES

  LITERAL DESCRIPTOR TABLE:     0 BYTES.      0 ENTRIES
  PICTURE DESCRIPTOR TABLE:     0 BYTES.      0 ENTRIES
KEYTABLE DESCRIPTOR TABLE:      4 BYTES.      1 ENTRIES
   FORMAT DESCRIPTOR TABLE:     0 BYTES.      0 ENTRIES

         LITERAL POOL SIZE:     0 BYTES
         PICTURE POOL SIZE:     0 BYTES
       KEYTABLE POOL SIZE:      4 BYTES
         FORMAT POOL SIZE:      0 BYTES

   INTERPRETABLE CODE SIZE:     0 BYTES
            PROGRAM LENGTH:    40 BYTES

          NUMBER OF ERRORS      0
```

2.3.3.11 *Segment map*

This map gives a listing of the number of segments, the number of modules contained in a segment and the number of bytes per segment. The format of the segment map is shown in the following example:

```
* CREDIT CODE LINKER PRR 4.1 790410 * SEGMENT MAP
```

SEGMENT NUMBER	SEGMENT TYPE	SEGMENT LENGTH	USAGE	NUMBER OF MODULES	NUMBER OF ERRORS
00	C	40		0	0
01	D	914	91 %	4	0

where: NUMBER is the segment number.
TYPE indicates:
C = common part (segment zero)
R = main memory resident
␣ = disk resident

LENGTH	number of bytes contained in this segment (program length).
USAGE	a filling percentage of the segment, related to the size option in the TLK command.
MODULES	number of modules contained in the segment.
ERRORS	number of errors per segment.

2.3.3.12 *Address cross reference listing*

This listing provides a cross reference between statement/subroutine identifiers in the procedure division and the modules/segments in which they are referenced. The format of the address cross reference listing is shown in the following example:

```
* CREDIT CODE LINKER PRR 4.1 790410 * CROSS REFERENCE LISTING                    * DATE 2857   * PAGE

SYMBOL  TYPE  VALUE    SEG-DEFINED    REFERENCES

DISC    P     02 0090  02-MODUL6      01-MAIN    (1)
GO      B S   01 000E  01-MAIN        02-MODUL6 (1)    02-MODUL7 (1)    02-MODUL8 (1)
GTP1    P     01 0030  01-MOD2        01-MAIN    (1)
GTP2    P     01 0046  01-MOD2        01-MAIN    (1)
GTP3          01 004C  01-MOD2
KB1     P     02 000E  02-MOD3        01-MAIN    (1)
KB2     P     01 0052  01-MODUL5      01-MAIN    (1)
T:ASSI  C                             01-MAIN    (1)
T:DSC1  C                             01-MODUL5 (8)    02-MOD3   (4)    02-MODUL4 (3)    02-MODUL6 (1)
                                      02-MODUL7 (1)
T:EDWR  C                             01-MOD1    (1)    01-MOD2   (3)    01-MODUL5 (4)    02-MOD3   (1)
                                      02-MODUL4 (1)    02-MODUL6 (1)    02-MODUL7 (1)    02-MODUL8 (1)
T:KI    C                             01-MAIN    (1)    01-MODUL5 (3)    02-MOD3   (2)
T:NKI   C                             01-MODUL5 (2)
T:RREA  C                             01-MODUL5 (1)
T:RWRI  C                             02-MODUL6 (1)
VDU1    P     01 0037  01-MOD1        01-MAIN    (1)
VDU2    P     02 006D  02-MODUL4      01-MAIN    (1)
VDU3    P     02 00B9  02-MODUL7      01-MAIN    (1)
VDU4    P     02 00C8  02-MODUL8      01-MAIN    (1)
```

where:	SYMBOL	is the statement/subroutine identifier in the procedure division.
	TYPE	indicates type of instruction in which "symbol" is used. C = CALL P = Perform B = Branch S = Start point.
	VALUE	displacement of "symbol" in the referenced segment. The first two digits specify the segment number and the next four specify the displacement within this segment.

SEG-DEFINED REFERENCES	segment number and module name which contains "symbol". are the segment numbers and module names, containing references to "symbol". The number of references in each module appears in brackets after the module name.

2.3.3.13 *Linker statistics total*

The format of the linker statistics total is shown in the following example. The contents of the listing are self-explanatory.

```
------------------------------------------------------------------------
* CREDIT CODE LINKER PRR 4.1 790410 * LINKER STATISTICS TOTAL
------------------------------------------------------------------------


  ALL VALUES DECIMAL

   INTERPRETABLE CODE SIZE:    547 BYTES
            PROGRAM LENGTH:   1568 BYTES

       AVAILABLE WORKSPACE:  23874 BYTES
            USED WORKSPACE:   2934 BYTES,     12 %
          UNUSED WORKSPACE:  20940 BYTES
  MAX WORKSPACE PER MODULE:    370 BYTES

           NUMBER OF ERRORS       0

PROG ELAPSED TIME:  00H-02M-01S-520MS-
```

3 TOSS SYSTEM START

3.1 General

System start procedure is the initialisation of a PTS Terminal computer for running an application program.
System start procedure comprises the following steps:

- Load the TOSS Monitor into memory
- Load the application into memory
- Read the configuration file and set up the required tables and buffers
- Pass control to the application or to CREBUG if this was included

The TOSS Monitor for a particular Terminal System must have been generated by the DOS utility SYSGEN. This results in a Monitor load module on disk.
Translator, CREDIT linker and linkage editor create an application load module on disk.
Generation of a configuration file is described in section 3.4.6 and 3.4.7.
These modules can then be copied to one or more cassettes by the DOS catalogued procedure $PCAS, or to TOSS formatted disk or flexible disk by $PDISC.

The use of SYSGEN, CONGEN, $PCAS and $PDISC is described in the DOS6800 System Software PRM M11.

3.1.1 *System load program SYSLOD*

To make the most economical use of core storage, a dynamic configuration procedure is included in the system software. This procedure SYSLOD is a software module which is linked to the TOSS Monitor.
After loading of the Monitor by the initial program loader, control is passed to SYSLOD. SYSLOD then loads the application load module from disk, cassette or flexible disk. After that SYSLOD reads the configuration file, and performs monitor and application configuration.
SYSLOD then passes control to the application or to CREBUG.
Configuration by SYSLOD must always take place, it is not possible to load an already configurated Monitor.

3.1.2 *Monitor configuration*

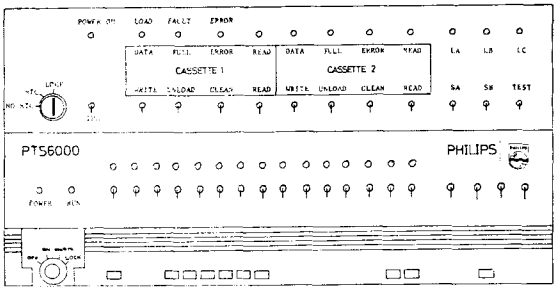
Monitor configuration is the process whereby the general Monitor created by SYSGEN is adapted to the specific hardware environment.
Monitor tables and buffers are built and the Monitor is supplied with pointers to the tables. In the case of PTS 6813 with MMU, the MMU tables are set up and an extra number of I/O buffers is reserved. Buffers for Data Communication and data management are also generated during system configuration.
If segmented application, the segment and page tables controlling the segments and pages in run time, are generated.

3.1.3 *Application configuration*

After the monitor configuration process, SYSLOD starts the application configuration. Terminal control areas, terminal work blocks, user work blocks, swappable work blocks, terminal stacks and data-set buffers are generated. In case of a system with MMU, the MMU tables are extended with references to the data division and the application. After application configuration, SYSLOD starts the system by queueing all tasks in the dispatcher queue and gives control to the interpreter.

3.2 Loading procedures

Monitor, application load module and configuration file can be loaded from cassette, disk or flexible disk.
Application and configuration data should reside on the same device type. If loading from disk, Monitor and application load module should be on the same volume.
Loading is controlled from the SOP or from the full panel and SOP if present.



System Operator's Panel

Full Panel

3.2.1 *SOP lamps*

During loading, the following SOP lamps indicate:

- SOP lamp 1 an application is loaded by SYSLOD
- SOP lamp 2 input error
- SOP lamp 3 memory overflow
- SOP lamp 4 format error
- SOP lamp 5 terminal ident error
- SOP lamp 6 user-or swappable work block error
- SOP lamp 7 MMU table overflow
- SOP lamp 8 illegal page size

3.2.2 *Program loading from cassette*

Monitor, application and configuration data can be loaded from the same or from different cassettes.

The procedure for loading from cassette is:

- Ensure that power is switched on at the Terminal computer and at each peripheral device.
- Ensure that the real time clock is on.
- Place Monitor cassette in a cassette drive.
- Press SOP switch IPL
  or where a full panel, press RST, MC, IPL in that order.
- Press SOP switch 1 for the left-hand cassette drive
  or SOP switch 2 for the right-hand cassette drive.
- Wait for loading indicator SOP lamp 1 to turn off.
  If the next module to be loaded is not on the same cassette, this cassette is now unloaded. Place the next cassette in a cassette drive and press the corresponding SOP switch.

– Turn key to Lock position.
– When loading is completed the application or CREBUG will be started.

3.2.3 *Program loading from disk*

Monitor and application must be loaded from the same volume.
The configuration file may be on the same or on a different disk or on flexible disk.

The procedure for loading from disk is:

– Ensure that power is switched on at the Terminal computer and at each peripheral device.
– Ensure that the real time clock is on.
– Place the Monitor disk on a disk drive, press the START button and wait for the READY lamp to light up.
If loading from the fixed disk, a disk cartridge must still be mounted.
– Press SOP switch IPL
or where a full panel, press RST, MC, IPL on the full panel.
– Press SOP switch 3 for loading from fixed disk,
or SOP switch 4 for loading from cartridge disk.
– If there is more than one application program on the disk, all SOP lamps will light up. Select the application to be loaded by pressing the corresponding SOP switch. Monitor and application are now loaded.
If during system generation (SYSGEN) was specified that there is only one application program on the disk, Monitor and application are now loaded automatically.
– If the configuration file is on the same volume as the Monitor and the application, SYSLOD automatically reads in the configuration data.
– If the configuration file is not on the same volume as the Monitor and the application, the eight leftmost SOP lamps will light up. Press the SOP switch corresponding to the device from which the configuration file must be read:

SOP switch 3	fixed disk
SOP switch 4	cartridge disk
SOP switch 5	flexible disk 0 multiplex channel
SOP switch 6	flexible disk 1 multiplex channel
SOP switch 7	flexible disk 0 programmed channel
SOP switch 8	flexible disk 1 programmed channel

– Turn key to Lock position.
– When loading is completed the application or CREBUG will be started.

3.2.4 *Program loading from flexible disk*

Monitor and application must be loaded from the same volume.
The configuration file may be on the same volume or on a fixed or cartridge disk.

The procedure for loading from flexible disk is:

– Ensure that power is switched on at the Terminal computer and at each peripheral device.
– Ensure that the real time clock is on.
– Place the flexible disk in disk drive 0 or 1 and wait for the READY lamp to light up.

- Press SOP switch IPL
  or where a full panel, press RST, MC, IPL on the full panel in that order.
- Select the flexible disk drive from which is to be loaded by pressing the corresponding SOP switch:

  SOP switch 5 flexible disk 0 multiplex channel
  SOP switch 6 flexible disk 1 multiplex channel
  SOP switch 7 flexible disk 0 programmed channel
  SOP switch 8 flexible disk 1 programmed channel

- If there is more than one application load module on the flexible disk, all the SOP lamps will light up. Select the application to be loaded by pressing the corresponding SOP switch. Monitor and application are now loaded.
  If during system generation was specified that there is only one application on the flexible disk, Monitor and application are loaded automatically.
- If the configuration file is not on the same volume as the Monitor and the application, the eight leftmost SOP lamps will light up. Press the SOP switch corresponding to the device from which the configuration file must be read:

  SOP switch 3 fixed disk
  SOP switch 4 cartridge disk
  SOP switch 5 flexible disk 0 multiplex channel
  SOP switch 6 flexible disk 1 multiplex channel
  SOP switch 7 flexible disk 0 programmed channel
  SOP switch 8 flexible disk 1 programmed channel

- Turn key to Lock position.
- When loading is completed the application or CREBUG will be started.

3.3 Program file layout

3.3.1 *Program file on cassette*

A cassette with the TOSS Monitor load module, the application load module and the configuration file is created with the DOS utility $PCAS. Application data may optionally be put behind these modules on the same cassette.

Layout of the program cassette:

TM	IPL MONITOR SYSLOD	TM	APPLICATION	TM	CONGIF FILE	TM	APPL DATA	TMTM

The different modules may also be on different cassettes. The loading program SYSLOD checks after every module if there is a double tape mark (TM).
In that case the cassette is unloaded and loading continues when the next cassette has been mounted and the corresponding SOP switch is pressed.
If no double tape mark is found after a module, loading continues from the same cassette.
Application data is not read by SYSLOD but has to be read by the application itself.

3.3.2 *Program file on disk*

A disk or flexible disk containing the TOSS Monitor load module, the application load module and the configuration file is created with the DOS utility $PDISC.
The disk or flexible disk must be TOSS formatted. The load modules should be on an L file, configuration data and, optional, application data on an S file.
The configuration file need not be on the same volume as Monitor and application.

3.4 Configuration file

SYSLOD performs configuration at load time, according to configuration data for the specific environment. These data are read from a configuration file which was first created under DOS.
The configuration file can be copied to cassette by $PCAS or to an S file on a TOSS formatted disk by $PDISC.
The configuration file should be on the same medium type as the application.

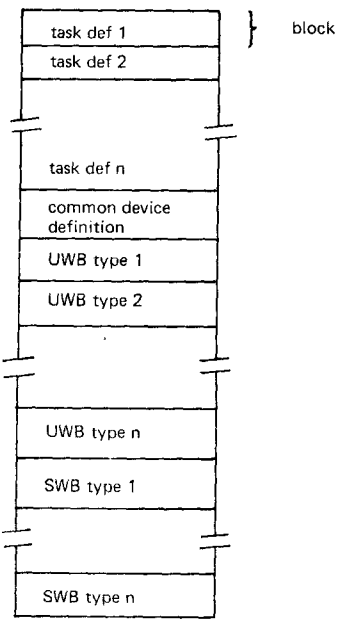
Requirements for the configuration file on disk:

- record length 9 and blocking factor 40 must be specified during CRF.
- File name should answer the format:

  $xxxx:nn
  where: x = alfanumeric character
  nn = 2 numerics representing the SOP switch number of the application (01 to 10).

The configuration file can also be created as an UF file on disk by standard DOS utilities and then moved to the load medium by $PCAS or $PDISC.

3.4.1 *Configuration file layout:*



3.4.2 *Task definition block layout*

Task definition block should have the following format:

DESCRIPTION	FORMAT	REMARKS	EXAMPLE
Block type	T;	Task block	T;
Number of tasks	nn;		06;
Task ID start value	TID=xn;		TID=F3;
Terminal class	TCL=xn;		TCL=F3;
Task level	LEV=nn;		LEV=60;
Number of terminal device classes	nn;	00 if no terminal device classes	03;
Terminal device class	TCD=Tnn;		TDC=T01;
Line connection	LC=nn{L/R};	number of channel unit, local or remote	LC=12L;
	TDC=Tnn;	repeat TDC and LC for each term. dev. class in the task	TDC=T02; LC=02R;
	LC=nn{L/R};		TDC=T03; LC=06L;
Number of special device classes	nn;	00 if no special device classes	02;
Special device class	SDC=Snn;	repeat SDC for each special dev. class in the task	SDC=S01;
			SDC=S02;

x = alphabetic character
n = numeric

3.4.3 *Common device definition block layout*

DESCRIPTION	FORMAT	REMARKS	EXAMPLE
Block type	C;	Common block	C;
Number of terminal device classes	nn;	00 if no terminal device classes	02;
Terminal device class	TDC=Tnn;		TDC=T02;
Line connection	LC=nn L; R	number of channel unit, local or remote	LC=03L;
	TDC=Tnn;		TDC=T03;
	LC=nn L; R	repeat TDC and LC for each term. dev. class.	LC=02R;
Number of special device classes	nn;	00 if no special device classes	03;
Special device class	SCL=Snn;	repeat SCL for each special dev. class	SCL=S04; SCL=S05; SCL=S06;

3.4.4 *User work block type definition block layout*

DESCRIPTION	FORMAT	REMARKS	EXAMPLE
Block type	U;	User work block	U;
Number of UWB types	nnn;		003;
Name of UWB	xxx;		UB1;
Number of blocks	nnn;	Number of blocks of this type	003;
		Repeat name and number for each UWB type	UC1; 005; UD1; 002;

3.4.5 *Swappable work block type definition block layout*

DESCRIPTION	FORMAT	REMARKS	EXAMPLE
Block type	S;	Swappable work block	S;
Number of SWB types	nnn;		002;
Name of SWB	xxx;		SB1;
Number of blocks	nnn,	Number of blocks of this type	003;
		Terminal name and number for each SWB type	SC1;
			004;

nn; number of tasks, in the current terminal class.

TID=xn; task identifier
Within every task definition block the first task has the specified ID, the next task has the specified value added by one and so on.
The CREDIT debugger must have a TID=TB.

TCL=xn; Terminal class to match, relates to the task identifier defined in the TERM statement in the data division. When TID and TCL have different names, the monitor task tables and application terminal control areas are configured with task identifiers according to the TID start value. TCL indicates that the current value in the TERM statement will not be used, when different from the start value in TID.

LEV=nn; priority level of the task is defined. Application tasks run normally on priority level 60, only the CREDIT debugger runs on a higher priority, level 55.

nn; number of terminal device classes used within this terminal class. Terminal device classes are defined at system generation time (SYSGEN). Devices contained in these terminal device classes are connected to a channel unit local terminals (CHLT) or channel unit remote terminals (CHRT).

TDC=Tnn; the terminal device class name as defined at system generation time, must be specified.

LC=nn L; line connection, local (L) or remote (R) to which the devices mentioned in the previous terminal device class (TDC=Tnn) are connected. Line connections are also added by one for each new task within this terminal class.

nn; number of special device classes used within this terminal class. Special device classes are defined at system generation time (SYSGEN). These devices are not connected to a channel unit local (CHLT) or channel unit remote (CHRT).

SDC-Snn; special device class name as defined at system generation time (SYSGEN), must be specified.

C; is the start of the common device definition block.

. nn; number of terminal device classes used common by all terminal classes.

. TDC=Tnn; the terminal device class name as defined at system generation time, must be specified.

. LC=nn L; line connection, local (L) or remote (R), to which the devices mentioned in the previous terminal device class (TDC=Tnn) are connected. Line connections are also added by one for each new task within this terminal class.

nn; number of special device classes containing devices to be used as common device. (For all tasks in the application.)
SCL-Snn; the special device class name as defined at system generation time (SYSGEN) must be specified.
U; is the start of the user work block definition block.
nnn; number of different user work block types.
xxx; name of the user work block, corresponding to the one defined in the data division.
nnn; number of copies wanted, of the previously defined user work block.
S; is the start of the swappable work block definition.
nnn; number of different swappable work blocks types.
xxx; name of the swappable work block, corresponding to the one defined in the data division.
nnn; number of copies wanted, of the previous defined swappable work block.

The line connections are also added by one for each new task within a group. The terminal device class ID and the special device class ID are specified by SYSGEN.

3.4.6 *Generating a configuration file on cassette*

The procedure for generating a configuration file on cassette without using $PCAS is shown in the following example:

First insert a cassette in one of the cassette drives, then key in the following sequence:

```
(i)    ASG   /E1,TY10
(ii)   RDA   /0A
(iii)  T;
       02;
       TID=B0;
       TCL=B0;
       LEV=60;
       02;
       TDC=T01;
       LC=1L;
       TDC=T02;
       LC=4L;
       01;
       SDC=S01;
(iv)   C;
       00;
       02;
       SCL=S02;
       SCL=S03;
(v)    U;
       002;
       UB1;
       004;
       UC1;
       005;
(vi)   S;
       001;
       SB1;
       005;
(vii)  :EOF
(viii) REW   /03
(ix)   WEF   /03
(x)    PCH   /0A
(xi)   WEF   /03
(xii)  ULD   /03
```

Explanation:

(i) Assign file code E1 (source input) to console typewriter.
(ii) Read data from the source input device (typewriter) and transfer to temporary disk file /0A.
(iii) Key in data for the task definition block. (May be repeated for other terminal classes.)
(iv) Key in data for the common device definition block.
(v) Key in data for the user work block type definition block.
(vi) Key in data for the swappable work block type definition block.
(vii) End of the read data command.
(viii) Rewind the cassette (unit number 1).
(ix) Write a tape mark on the cassette.
(x) Write the contents of the temporary disk file /0A to the cassette (inclusive a tape mark).
(xi) Write a second tape mark on the cassette.
(xii) Unload the cassette.

3.4.7 *Generation of a configuration file on a DOS disk*

The procedure for generating a configuration file on a DOS disk is shown in the following example. With the catalogued procedure $PCAS the configuration file can be copied to cassette.

```
(i)   ASG   /E1,TY10
(ii)  RDA   /20
(iii) T;
      02;
      TID=B0;
      TCL=B0;
      LEV=60;
      02;
      TDC=T01;
      LC=1L;
      TDC=T02;
      LC=4L;
      01;
      SDC=S01;
(iv)  C;
      00;
      02;
      SCL=S02;
      SCL=03;
(v)   U;
      002;
      UB1;
      004;
      UC1;
      005;
(vi)  S;
      001;
      SB1;
      005;
(vii) :EOF
(viii) KPF /20,CONFIG
(ix)  $PCAS C=CONFIG
```

Explanation:

(i) Assign file code E1 (cource input) to console typewriter.
(ii) Read data from the source input device (typewriter) and transfer to temporary disk file /20.

(iii) Key in data for the task definition block.
(May be repeated for other terminal classes.)
(iv) Key in data for the common device definition block.
(v) Key in data for the user work block type definition block.
(vi) Key in data for the swappable work block type definition block.
(vii) End of the read data command.
(viii) Keep the file as library file.
(ix) With $PCAS the configuration data can be written to cassette

(For details see PRM DOS6800.)

3.4.7 *Errors during system run time*

For indicating the type of error, causing a program halt, the following SOP lamps are lit before the execution of the program is stopped.
SOP lamps are numbered from 1–11, number 1 being the left-most lamp.

SOP lamps lit					Significance
7	8	9	10	11	
			x	x	No currency buffer available
		x		x	Illegal interrupt
		x	x	x	Stack overflow
	x			x	Instruction not accepted SST, OTR or INR not accepted due to hardware error
	x		x	x	No blocks available
	x	x		x	Invalid instruction (trap)
	x	x	x	x	Requested LKM processor missing
x				x	Data management (SYSGEN) error

4 CREDIT DEBUGGING PROGRAM

4.1 Introduction

The CREDIT debugging program (CREBUG) is an interactive diagnostic task which runs under the control of the TOSS Monitor, on systems with and without memory management. It runs in parallel with a CREDIT application program being tested.

CREBUG may be used to control the execution of the application program in the following ways:

- Traps may be inserted.
- Verification may be started and stopped.
- The application program may be started or stopped.
- Variables may be examined and modified.
- Trace may be turned on or off.

In addition, CREBUG may be used to:

- Perform calculations (e.g. on addresses).
- Dump Memory.

Readers of Section 4 should be familiar with the following DOS6800 System Software concepts:

- Linkage editor
- Control command
- User library
- TOSS system operation

These concepts are explained in the DOS6800 System Software PRM (M11).

4.2 Running CREBUG

CREBUG is automatically built into the application load module by the Linkage Editor. If CREBUG is not required (e.g. for production versions of the application program) it must be explicitly excluded. This procedure is described below.
CREBUG has a task identity of TB and runs at priority level 55, specified in the configuration file for system loading (SYSLOD). The CREDIT interpreter calls CREBUG immediately before executing each instruction in the application program. The application program status is then checked. If certain conditions specified by the programmer are met, either the program is stopped or specified memory locations are printed.
The programmer communicates with CREBUG via one of the following device configurations:

General printer PTS6321, together with alphanumeric keyboard PTS6234 or 6331.
Console typewriter PTS6862.
Visual display unit PTS6344.

The chosen device configuration must be assigned TOSS file code /21 for input and /31 for output. File code /16 must be assigned to a (line) printer, when tracing is used. This is done during TOSS system generation.

The memory size of the application program is increased when CREBUG is included. For this reason it is advisable to explicitly exclude CREBUG when linkage editing the production version of the program. The following command sequence is recommended:

```
USERID : X
S:INC␣/OBJCT,INT:PROD
S:KPF/O
SCR
S:INC␣/OBJCT,Y
KPF/O
MOV␣MAIN,/S,USER
S:KPF /S,MAIN
S:TRA MAIN,NL
S:TLK␣MIX
S:LKE␣U,M
S:KPF␣/L,MOD NAM
```

where: X is an empty user library and Y is the user library containing the CREDIT Linker object modules.
MAIN is the module containing the data-division.

4.3 CREBUG Input

4.3.1 *General*

Various single character commands may be keyed-in by the programmer to control the testing process. These commands are:

T Set trap
P Proceed from trap
L Loop through trap
Y Remove trap
V Start or stop verification
G Go
H Halt
I Open data item
J Open boolean data item
Q Open Relocation Register
S Open Task Control Area/Condition Register
W Open memory word
/ Open byte
R Turn trace on or off
= Calculate
M Dump memory
KL Lock segment
KU Unlock segment

4.3.2 *CREBUG Modes*

CREBUG operates in one of two modes known as T mode and U mode. In T mode CREBUG is running and the application program is stopped. In U mode both CREBUG and the application program are running. T mode is selected whenever the application program stops.

A stop occurs when:

- a halt (H) command is keyed-in, or
- a trap is encountered in the application program, or
- a verification halt condition is detected.

U mode is selected when one of the following commands is keyed-in:

- proceed from trap (P)
- loop through trap (L)
- go (G)

Commands other than H, P, L and G will not result in a change of mode. The current mode is indicated by the letter T or U printed at the left of each line of output. Immediately after System start CREBUG is in T mode.

4.3.3 *Current Task Identifier/Current Segment number*

In certain commands a task identifier/segment number may be specified. If no task identifier or segment number is specified in these commands, the 'current' task identifier/segment number is assumed. This is the identifier of the task or segment number which was executed when the program halted last.

4.3.4 *Relocation register*

CREBUG maintains 16 relocation registers for the whole application, also when memory management is used. The registers are numbered 0 to F and may be used in commands as indices when referring to memory locations in the segments. The contents of relocation registers may be examined and modified using the Q command. The following description illustrates the way in which relocation registers are normally used. The start address of the module, currently being tested, and its segment number are loaded into a relocation register. Commands then refer to locations within this module by quoting the address relative to the start of the module, listed by the translator under the heading LOC, together with the segment number. Loading relocation register 4 with the start address of the module, which is present in segment 1, is done as follows:

4Q/0000 11E.1

A trap in this module and segment is set as : 72, 4T.
For non segmented program in memory always segment number zero must be specified.
When starting an application, relocation register 0 contains the start address of the program code (P:PIL), but this register is not used when setting traps. Program code is found by using the segment number and for non segmented applications, segment zero is specified. Register F contains minus eight (X'FFF8') which can be used to bias addresses from the linkage editor map.

4.3.4 *Addressing*

The following commands contain references to memory addresses:

I,J
G,P,T,Y
M,V,W,/

Either relative or absolute addresses may be used in these commands.
If an absolute address is used it may be written in either of the following ways:

hexadecimal-number[index]
$decimal-number[index]

The absolute address of a memory word is its displacement from word zero of memory. If the address refers to the start of an array, an index value must be given to specify the word within the array (counting the first word as one).

If relative address is used it may be written in the above manner. But if the relative address refers to a word in a procedure division, it must be modified by the appropriate segment number as follows:

hexadecimal-number[index].segmentnumber
$decimal-number[index].segmentnumber

The relative address of a memory word is its displacement from word zero of the CREDIT module of which it forms a part. Relative addresses are shown in a CREDIT module under the heading LOC (Location counter).

Commands I and J may only refer to addresses in the data division.
Commands G,P,T and Y may only refer to addresses in the procedure division.
Commands M,V, W and / may refer to addresses in either division.

In order to differentiate between the index and the location counter generated by the CREDIT translator, addresses referring to the data division must be prefixed by a # character (e.g. #18W).

Indirect addressing may be used in the following open variable commands:

Open data item (I)
Open task control area/condition register (S)
Open memory word (W)

If an address is indirect it must be prefixed by an asterisk (e.g. *10W). In case the address will point to a memory word which contains the address of the variable to be opened. If the resulting address is odd the next (lower) even address will be used.

4.3.6 *Command syntax*

Commands are keyed-in immediately after the T or U prompt. The prompt is printed at the left of each line by CREBUG.

Commands have the following syntax:

[arg1;] [arg2] [:{tid|pha}] [.segnr] com

com is one of the single character commands listed above.
tid is the task identifier of the task to which the command applies. Task identifiers are defined during system loading time (SYSLOD).
segnr is the segment number.
pha is a physical memory area. The current user is default. (Only with MMU systems.) pha may be defined as :S, :X, :Y or :Z. Each value assigns a specific physical memory area.

:S System area, 0–64Kb
:X Extended area 64Kb–128Kb
:Y Extended area 128Kb–192Kb
:Z Extended area 192Kb–256Kb

Arg1 and arg2 are defined as follows:

arg1 :: = { term / terms }

arg2 :: = arg1

term :: = [{ # / * }] address[index] [,relocation-register]

terms :: = term{ + / − }term

index :: = ({ hexadecimal-integer / $decimal-integer } [,hexadecimal-integer]) $decimal-integer

address :: = { hexadecimal-integer / $decimal-integer }

relocation-register :: = 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

The following words, used in the above syntax definition, have the same meaning as that given in Appendix 1.

decimal-integer
hexadecimal-digit
hexadecimal-integer

4.4 CREBUG Output

4.4.1 *Program Stop Message*

Whenever the application program stops CREBUG prints the following message:

c pp=loc,rel.segnr:tid

The significance of these fields is as follows:

c — This code indicates why the program stopped. It may have the following values:
- S — System start is complete and the application program may be started.
- T — A trap has been encountered.
- H — An H (halt) command has been keyed in.
- V — A verification halt condition has been detected.
- E — The application program is in error.

pp — This is the value of the program pointer. It points to the interpretive instruction which will be obeyed when the program is restarted.

loc — This is the location counter value for the instruction pointed to by the pp. It is the value found under the heading "LOC" for this instruction in the CREDIT module listing. This value, when added to the contents of the relevant relocation register, gives the value of the program pointer. If no relocation register is found for the current segment "loc,rel" will be replaced by the word "SPACE".

rel — This is the relevant relocation register. When added to "LOC" it gives the value of the pp.

segnr — The segment number in which the program is halted. for non segmented programs, segment number zero is printed.

tid — This is the task identifier. It is the identifier of the application task which is currently executed.

4.4.2 *Command Responses*

The response to commands I, J, Q, S, W, / and = is printed immediately after the command on the same line. For example, open relocation register (Q):

1Q/0000 0006.2 (CR)

In this example, the programmer keys in the underlined characters and CREBUG prints the others.

The response to the other commands, appears on the line after the command. For example, dump memory (M):

0004.0M (CR)
208C 0153 1081 3002 831C 0053 2181 4910 4000

If an undefined command is keyed-in, CREBUG responds with a question mark and no action is taken.

If an illegal command is keyed-in, CREBUG responds with "NO!". An example of an illegal command is a go (G) command issued while the application program is actually running.

4.4.3 *Curtail Printout*

If the output from a CREBUG command is unexpectedly long (e.g. a large memory dump) it can be curtailed by depressing any SOP switch or by switching the terminal computer power off and on.
CREBUG uses file code /14 for SOP input, which can be included at system generation time.

4.4.4 *Error messages*

One of the following messages may be output, when the CREDIT debugger detects an error:

ILLEGAL OPERATION
MIXED ARITHMETIC
ILLEGAL TYPE
ODD WORD ADDRESS
MISSING ENTRY
ILLEGAL REQUESTED LENGTH
STACK UNDERFLOW
ARITHMETIC OVERFLOW
STACK OVERFLOW
INDEX OVERFLOW
ILL INDEX TYPE
DEV BY ZERO
EDIT BUFFER OVFL
EDIT PICTURE OVERFLOW
ALLOCATION ERR AT INITIALIZATION
ILLEGAL INSTRUCTION ADDRESS
ILLEGAL FORMAT CODE
NO FIX BUFFER ALLOCATED
FIX BUFFER NOT ALLOWED
ILLEGAL CONTROL CODE
ILLEGAL INDEX VALUE
FORMAT CONDITIONS CHANGED
ILLEGAL PARAMETER
LENGTH ERROR
DATASET BUSY
ILLEGAL DATASET REFERENCE
DISK ERROR NO REENTER POINT

4.4.5 *Trap Commands*

In order to examine the state of a program at a certain stage in the execution, the programmer may insert traps. A trap is an address specified to CREBUG at which the program is stopped. A stop message (described above) is printed by CREBUG and the system then waits for a further command from the programmer.

The instruction on which the trap is set is not executed until the program is re-started. However, it is possible to loop a number of times through a certain trap before the program is stopped.

A maximum of 16 traps at a time may be set.

4.4.6 *Open variable commands*

These commands are:

- Open data item (I)
- Open boolean data item (J)
- Open relocation register (Q)
- Open task control area/condition register (S)
- Open memory word (W)
- Open byte (.)

An open and modify command has the following form:

...../xxxx[arg1[: tid|pha]] [.segnr] com
xxxx is the contents which will be replaced optionally by arg1.

The function of these commands is to print the contents of the specified variable, and if requested, modify those contents. This is done in the following manner:

- the programmer specifies the variable and keys in one of the above commands. This is done according to the normal command syntax rules.
- CREBUG responds by printing the contents of the variable (/xxxx). The variable is now "open", i.e. it can be modified.
- If necessary the programmer can now key-in a new value for the variable. If the variable is not to be modified, the programmer simply keys in an LF or CR character and the variable is "closed".

Where possible the above dialogue is all printed on a single line, for example:

EQ/0000 2942,2 (CR)

In this example the programmer keys in the underlined characters and CREBUG prints the others.
"E" indicates that relocation register E is to be opened (command Q). CREBUG responds by printing out "/" followed by the contents of relocation register E "0000". The programmer then keys in the new value "2942" followed by the segment number and a carriage return character. The CR indicates that the variable is to be closed.

If a line feed character (LF) had been keyed-in instead of CR in the above example, it would indicate that the current variable is to be closed and that the variable at the next (higher) address is to be opened.
If an @ character has been keyed-in, it would indicate that the current variable is to be closed and used as the address of the next variable to be opened (i.e. indirect addressing). This variable will be automatically opened. The @ character is meaningful only in commands I, S and W.

Any other non-hexadecimal character in this position would result in the current variable being closed without modification and without opening the next variable.

If the programmer does not wish to modify the contents of the opened variable, it can be closed without modification by simply keying-in a CR or LF character immediately after the current variable contents.

The possible values of the terminating character are summarised as follows:

- CR – close current variable
- LF – close current variable and open next variable
- @ – close current variable and open indirectly addressed variable.

4.4.7 *Command reference*

G	*Go*	G

Syntax: (i) arg2 [.segnr] G
(ii) G

Description: A go command enables the programmer to start an application program which has stopped.

(i) Start the program at address "arg2". Current segment is default.
(ii) Start the program at the address pointed to by the program pointer.

Example:

0005G	start at location 5 in segment 0
0007.3G	start at location 7 in segment 3
0004,2.4G	start at location 4 modified with relocation register 2, in segment 4.

H *Halt* H

Syntax: (i) :*tid* H
(ii) H

Description: (i) Halt the task specified by "tid".
(ii) Halt the program (may be any task).

Example: :BOH halt for task "BO"
H halt program

Open data Item

Syntax: arg2 [:tid] I

Description: This command can be used with binary, decimal or string data items. If the item is binary it will be opened. If it is decimal or string, it will be printed but not opened. The current "tid" is default. To open string or decimal data items, the "IX" value from the listing, must be preceded by a # sign, and followed by "W".

Example:

37(3)I/0005	Open element 3 of the array, referenced in the listing by 37.
50I/0002	Open binary data item referenced by 50, for current task.
#31W/2020	Open string data item referenced by 31
#42W/B137	Open decimal data item referenced by 42
50I:B0I/0521	Open binary data item referenced by 50, for task B0.

J — *Open boolean data item* — J

Syntax: arg2 [:tid] J

Description: This command is used to open a boolean data item. The current "tid" is default.

Example:
10J/0 Open boolean data item referenced by 10 for the current task.
10J:B0J Open boolean data item referenced by 10 for task B0.

KL *Lock segment* **KL**

Syntax: arg2 KL

Description: The segment specified in arg2 becomes main memory resident. It stays resident until an unlock segment (KU) command is executed. The command KL can be used before making a patch in a segment.

Example: 2KL Segment 2 becomes main memory resident.

KU *Unlock segment* **KU**

Syntax: arg2KU

Description: The segment with the number specified in arg2 will no longer be main memory resident.

Example: 2KU Segment 2 released from main memory.

L — *Loop through trap* — L

Syntax: (i) arg2 L
(ii) L

Description: (i) Loop "arg2" times through the current trap.
(ii) Loop once through the current trap.

Note: This command can only be given when the application program is stopped at a trap.

Example: 5L loop 5 times through current trap
L loop once through current trap

M — *Dump memory* — M

Syntax:
(i) arg1; arg2 [:{tid|alpha}] [.segnr] M
(ii) arg2 [:{tid|alpha}] [.segnr] M
(iii) M

Description: The contents of selected memory words can be printed. When one or more lines have been printed CR (carriage return) or LF (linefeed) may be given. If CR is keyed-in the command is terminated. If LF is keyed-in the next eight memory words will be printed.

(i) Dump memory words from address "arg1" to address "arg2" inclusive. Current "tid" is default.
(ii) Dump eight memory words from address "arg2". Current "tid" is default.
(iii) Dump eight memory words from the address of the last word dumped.

Examples:

0004; 0018 M	Dump memory words 4 to 18 of the user area of the current task
0004; 0018:B0M	Dump memory words 4 to 18 of the user area of task B0
0004.5; 0018.5M	Dump memory words 4 to 18 of segment 5
0004; 0018::$M	Dump memory words 4 to 18 in the System area.

P *Proceed from trap* **P**

Syntax:
(i) arg1; arg2 [:tid] [.segnr] P
(ii) arg2 [:tid] [.segnr] P
(iii) P

Description:
(i) Set a new trap at address "arg1" and set the loop counter to "arg2". Remove current trap and proceed with program execution. Current segment is default.
(ii) Set a trap at address "arg2" and set the loop counter to zero. Remove current trap and proceed with program execution. Current segment is default.
(iii) Remove the current trap and proceed with program execution.

Note: This command can only be given when the application program is stopped at a trap.

Q — *Open relocation register* — Q

Syntax:
(i) arg2Q
(ii) Q

Description:
(i) Open relocation register "arg2". Only the last four bits are significant. 16 relocation registers are available, numbered from hexadecimal '0' to 'F'.
(ii) Print all relocation registers.

Example: 5Q/0000 ␣ 3D.0 Relocation register 5 is loaded with the start address of MOD2 (3D, from the load map), modified with the start address of segment 0.

R *Trace* R

Syntax: (i) arg2R
(ii) R

Description: This command is used to trace the execution sequence of some specific instructions or all. The trace mode is defined in arg2 and can be:

0 – switch trace off
1 – trace branch instructions
2 – trace all instructions
3 – trace arithmetic instructions.

(i) Set trace mode to "arg2". This affects all tasks.
(ii) Stop tracing.

S — *Open task control area / Condition register* — **S**

Syntax: (i) arg2 [,tid] S
(ii) [,tid] S

Description: This command can be used to open a word in the task control area (TCA) or to open a condition register.
The current "tid" is default.
The addresses of the items in the TCA are shown below.

	FCB (optional)	Format Control Block (33 words)
	DSCB's	Data set Control Blocks. 10 words per data set. The layout corresponds to an ECB.
−14	CSE	Current Segment End.
−12	CSB	Current Segment Base.
−10	CSN	Current Segment Number.
−8	TDAD	Pointer to task descriptor table.
−6	CIA	Current Instruction Address.
−4	TID	Task identifier.
−2	STKE	Stack end pointer.
0	PA	Auxiliary Stack pointer
+2	STKB	Stack Base pointer
+4		Descriptor table address.
+6		Work block address.

(i) Word "arg2" (hexadecimal) within the TCA is opened.
Current "tid" is default.

(ii) The condition register belonging to task identifier "tid" is opened.
"tid" is default.

Example: −4,A0/S — Open word −4 of TCA of task A0.
S/1 — Open condition register of current task
A0/S/2 — Open condition register of task A0.

T *Set trap* T

Syntax: (i) arg1; arg2 [:tid] [.segnr] T
(ii) arg2 [:tid] [.segnr] T

Description: (i) Set a trap at address "arg1" and set the loop counter to "arg2".
(ii) Set a trap at address "arg2" and set the loop counter to zero.

Note: If a "tid" is specified the trap affects only that task. If no "tid" is specified, the trap affects all tasks. Current segment is default.

V

Verify

V

Syntax:
(i) arg1, arg2 [:tid] V xx
(ii) arg2 [:tid] V
(iii) V

Description: The verify command instructs CREBUG to continuously monitor the contents of a specific memory word, referenced by arg1, during program execution. When the memory word assumes a specific relationship (equal to, greater etc.) to a given value, in arg2, the program is stopped.
A stop message is printed by CREBUG and the system waits for further command from the programmer. Verification is only permitted on complete 16 bit memory words.

(i) "XX" specifies the relationship to be tested for, and can have the following values:

EQ	meaning: (arg1)	= arg2
GR	meaning: (arg1)	> arg2
LT	meaning: (arg1)	< arg2
NE	meaning: (arg1)	≠ arg2
NG	meaning: (arg1)	≯ arg2
NL	meaning: (arg1)	≮ arg2

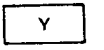
Stop program when the contents of "arg1" and the value in "arg2" meets the specified relationship in XX.
Current ' tid" is default.
(ii) Stop verification at "arg2",
(iii) Stop all verifications.

W *Open Memory Word* **W**

Syntax: (i) arg2 [:{tid | pha}] [.segnr] W
(ii) W

Description: (i) Open memory word at address "arg2". Current "tid" is default.
(ii) Reopen the memory word specified in the preceding W command.

Remove trap



Syntax: (i) arg2Y
(ii) Y

Description: (i) Remove the trap at address "arg2".
(ii) Remove all traps.

/ *Open Byte* /

Syntax: (i) arg2 [:tid] [.segnr] /
(ii) /

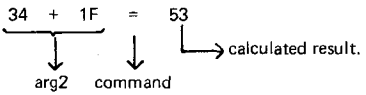
Description: (i) Open byte at address "arg2".
Current segment is default.
(ii) Re-open the byte specified in the preceding / command.

=	*Calculate*	=

Syntax: arg2=

Description: This command is used to calculate (add/subtract) with hexadecimal values.

Example:



APPENDIX A : CREDIT SYNTAX DEFINITION

This appendix defines the various syntactic items used in the foregoing instruction, directive and declaration syntax definitions. The symbols used below are explained in Section 1.1.

actual-parameter :: = { data-item-identifier | literal constant | array-identifier [,index-identifier-1] [,index-identifier-2] | format-list-identifier | formal parameter | key-table-identifier | data-set-identifier }

alphanumeric-character :: = { letter | decimal-digit }

array-identifier :: = identifier

array-type :: = { BCDI | BINI | STRGI }

bit :: = { 0 | 1 }

block-identifier :: = identifier

condition mask :: = 0|1|2|3|4|5|6|7

control-value :: = value Ivalue expression

data-item-identifier :: = { identifier | array-identifier(index-identifier-1 [,index-identifier-2]) | formal-parameter }

data-item-specification :: = { length [[value-type] ['value]] | length ['value'] | value-type ['value'] | 'value' }

data-item-type :: = { BCD | BIN | BOOL | STRG }

data-set-identifier :: = identifier

decimal-digit :: = 0|1|2|3|4|5|6|7|8|9

decimal-integer :: = decimal-digit . . .

decimal-number :: = [{ + | - }] decimal-integer

device-type :: = CR IDC IDI IDL IDN IDY IGP IIII IIO IKA IKI IKN ILP IMT ISI ISO ITK ITJ I TR ITV ITW I

dimension :: = decimal-integer

entry-identifier :: = identifier

equate-identifier ; ; = identifier

external-identifier :: = identifier

file-code = hexadecimal-digit hexdecimal-digit

file-name-identifier :: = { identifier | array-identifier(index-identifier-1 [,index-identifier-2]) | formal-parameter }

formal-parameter = { identifier ( ) [,identifier] | identifier (,) [,identifier,identifier] | identifier | $identifier }

format-list-identifier :: = identifier

format-table-identifier : : = identifier

hexadecimal-digits :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

hexadecimal-integer :: = hexadecimal-digit . . . 1

identifier :: = letter [alphanumeric-character] . . .] 8

index-identifier :: = identifier

key-table-identifier :: = identifier

key-value :: = { value-expression | equate-identifier }

label :: = identifier

length :: = decimal-integer

letter :: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

literal constant :: = [value-type] 'value'

module-name :: = identifier

picture-character :: = A | B | E | F | P | T | V | X | Z | 0 | 9 | + | − | * | . | ,

picture-string :: = 'picture-character . . .'

pointer-identifier :: = { identifier | array-identifier(index-identifier-1 [index-identifier-2]) | formal-parameter }

size-identifier :: = { identifier | array-identifier(index-identifier-1 [index-identifier-2]) | formal-parameter }

statement-identifier :: = identifier

string :: = string-character . . .

string-character :: = ISO-7-character

subroutine-identifier :: = identifier

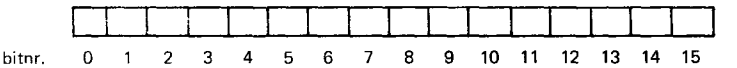
task-identifier :: = letter decimal-digit I letter letter

```
value :: = { decimal-number
           | hexadecimal-integer
           | string
           | decimal-integer }

value-expression :: = { decimal-integer
                      | value-type 'value'
                      | equate-identifier }
                      [ { + | - }
                        { decimal-integer
                        | value-type 'value'
                        | equate identifier } ... ]

value-type :: = C | D | W | X

volume-identifier :: = { identifier
                       | array-identifier(index-identifier-1
                                          [index-identifier-2] )
                       | formal-parameter }
```

APPENDIX B : EXTENDED STATUS CODES

This appendix explains the various status codes which may be returned to the application by the XSTAT instruction.
The general form of the extended status word is shown below. Some bits have a standard meaning.

Format of the status word:



Bit	Meaning
0	Illegal request
1–2	Not used/device dependent
3	End of file
4–8	Not used/device dependent
9	Timeout/Hardware error
10	Device dependent
11	Illegal order sequence
12	Incorrect length
13	Data fault (Parity/CRC/LRC/code)
14	Throughput error
15	Device not operable

Further information concerning standard bits and details of non standard bits are given for each device (driver) on the following pages.

The table below shows which driver belongs to which peripheral device.

Peripheral driver	Drivers
Card reader	DRCR01
Cassette recorder	DRTC01
Console typewriter	DRTW01
Data communication – HDLC multipoint	DRDC07
Data communication – BSC multipoint	DRDC15
Data communication – BSC point-to-point	DRDC17
Data communication – Univac 100 -200 synchronous	DRDC22
Data communication – SDLC master driver	DRDC81
Data communication – HDLC master driver	DRDC82
Disk	DRDU01
Flexible disk	DRFD01
General terminal printer	DRGP01
Intertask communication	DRIC01
Keyboards – general – port, PTS6236/6271/6272	DRKB01 DRKB03
Line Printer	DRLP01
Magnetic tape recorder	DRMT01
Signal displays, keyboard lamps	DRDI01
SOP	DRSOP1
Teller terminal printer	DRTP02, 3
Video and plasma displays	DRDY01

DRCR01 *Card reader* **DRCR01**

This status code is valid for a PTS6885 card reader.

The following bits may be set by this driver:

Bit	Meaning
0	Illegal request
3	End-of-File detected
10	Input hopper empty or Output stacker full
12	Incorrect length
13	Data fault
14	Throughput error
15	Not operable

Bit 12 is set if the requested number of characters is greater than 80, or if there is more information on the card than has been specified by the requested number of characters.

Bit 13 is set if a character is read, which cannot be converted.

Bit 14 is set if the card reader offers a new character, before the previous one has been taken care of by the driver.

Bit 15 card reader not operable (e.g. power off).

DRDC07 *Data communication HDLC multipoint data link* DRDC07

The following bits may be set by this driver:

Bit	Meaning	Read	Write	Transfer Parameters	Set Timeout
0	Illegal request	x	x	x	x
2	Status change	x			
9	Timeout/Poll timeout	x	x		
10	Carrier off	x	x		
14	Throughput error	x	x		
15	Modem not operable	x	x	x	

DRDC15 *DSC multipoint data communication* DRDC15

The following bits may be set by this driver:

Bit	Meaning	Read	Write	Transfer Parameters	Set Status	Set Time out
0	Illegal request	x	x	x	x	x
2	Status change	x				
5	Calling indicator	x				
9	Time out	x	x			
10	Carrier off	x				
13	Code check error	x	x			
14	Throughput error	x	x			
15	Not operable	x				

DRDC17 *DSC point-to-point data communication* DRDC17

The following bits may be set by this driver:

Bit	Meaning	Read	Write	RVI	Accept Call	Connect Line	Disconnect Line	Set Time out
4	WACK count out		x					
7	ETB received	x						
8	End of Transmission	x	x	x				
9	Time out*	x	x	x	x			
10	RVI received		x					
11	See below*		x	x	x			
12	Incorrect length	x		x				
14	Throughput error		x					
15	Not operable	x	x	x	x	x		

The reasons for these bits being set varies according to the instruction that was issued, as follows:

Bit 9 – Read
No message block has been received within the specified time.
– Write
This bit is set at unsuccessful DIB or when there has been no acknowledgement on a message block following several ENQ's
– RVI
No response has been received to WACK within a specified time.

Bit 11 – Write
ENQ received: this bit is set at BID collision or when the driver is in receive mode.
– RVI
Sequence error: set when the driver is in write mode or a block containing ETX has already been received.
– Accept call
Modem already connected.

DRDC22

Uniscope 100–200 synchronous data communication

DRDC22

The following bits may be set by this driver:

Bit	Meaning	Read	Write	Transfer Parameters	Set Status	Set Time Out
0	Illegal request	x	x	x	x	x
2	Status change	x	x			
5	Bell message received	x	x			
9	Time out/Poll Time out	x	x			
10	Carrier off	x	x			
14	Throughput error	x	x			
15	Modem not operable	x	x			

DRDI01 *Signal displays and lamps on keyboards* **DRDI01**

This status code is valid for signal displays PTS6241 and 6242, lamps on the keyboards PTS6232, 6233, 6234, 6236, 6271 and 6272 and the lamp functions of the badge card reader PTS6261.

The following bits may be set by this driver:

Bit	Meaning	Set Lamps On	Set Lamps Off	Flash Lamps
0	Illegal request	x	x	x
13	Code check error			
15	Not operable	x	x	x

DRDY01 *Video and plasma displays* **DRDY01**

This status code is valid for video display PTS6344, or the plasma displays PTS6351 or PTS6386 or the alphanumeric display PTS6385.
All alphanumeric characters within the range /20–/5F are sent from the buffer to the display, codes /60–/7F are reduced by /20, giving /40–/5F.

The following bits may be set by this driver:

Bit	Meaning	Test Status	Write	Set Cursor	Erase Line
0	Illegal request	x	x	x	x
13	Code check error		x	x	
14	Throughput error				
15	Not operable	x	x	x	x

DRGP01 *General printer* **DRGP01**

All alphanumeric characters in the range /20–/5F, in the user buffer, are sent to the printer. Codes /60–/7F are reduced by /20, giving /40–/5F.

The following bits may be set by this driver:

Bit	Meaning	Test Status	Write
0	Illegal request	x	x
13	Code check error		x
14	Throughput error		
15	Not operable	x	x

The following bits may be set by this driver:

Bit	Meaning	Read	Write	Random Read	Random Write	Set Time Out
0	Illegal request	x	x	x	x	x
9	Time out	x	x	x	x	
12	Incorrect length	x	x	x	x	

DRKB01 *Keyboard* **DRKB01**

This status code is valid for the keyboards PTS6231, 6232, 6233, 6234, 6331 and 6342 and the PTS6261 badge card reader with PIN keyboard.

The following bits may be set by this driver:

Bit	Meaning	Read	Skip Buffer
0	Illegal request	x	x
9	Time out	x	
12	Incorrect length	x	
13	Undefined key	x	
14	Throughput error	x	

Bit 14 is set if circular input buffer overflow occurs.
Bit 12 is set if overflow in the user buffer occurs.

DRKB03 *Keyboard* **DRKB03**

This status code is valid for the keyboards PTS6236, 6271 and 6272 and for the PTS6261 badge card reader with PIN keyboard.

The following bits may be set by this driver:

Bit	Meaning	Read	Skip Buffer
0	Illegal request	x	x
9	Time out	x	
12	Incorrect length	x	
13	Undefined key	x	
14	Throughput error	x	

Bit 12 is set if the user buffer overflow occurs.
Bit 14 is set if circular input buffer overflow occurs.

DRLP01 *Line printer* **DRLP01**

This status code is valid for the line printer PTS6881.
The following bits may be set by this driver:

Bit	Meaning	Test Status	Write
0	Illegal request	x	x
15	Not operable	x	x

DRMT01 *Magnetic tape* DRMT01

This status code is [illegible] 1/2" magnetic tape recorders PTS6872 or 6164.
The following [illegible] by this driver.

Bit	Meaning	Test Status	Rd	Write	Write Tape Mark	Rewind	Step Reverse	Step Forward	Load	Unload	Recover
0	Illegal request	x	x	x	x	x	x	x	x	x	x
2	Rewinding	x	x	x	x	x	x	x	x		
3	Tape mark		x		x		x	x			
4	No data		x	x	x		x	x			
5	BOT	x				x	x		x		x
6	Write protect	x	x	x	x	x	x	x	x		x
9	Hardware error		x	x	x	x	x	x	x	x	x
10	EOT	x	x	x	x		x	x			x
11	Sequence error		x								
12	Incorrect length		x								
13	Data error		x	x			x	x			x
14	Throughput error		x	x							
15	Not operable	x	x	x	x	x	x	x	x	x	x

Bit 4 is set if data [illegible] within two seconds.
Bit 11 is set if the block [illegible] counter is found incorrect (if in use).
Bit 12 is set if the [illegible] number of characters was less than the actual length.

Only bit zero of the extended status code is used, and this bit is set if any error is detected.

The following bits may be set by this driver:

Bit	Meaning	Test Status	Read	Write	Write Tape Mark	Erase	Lock	Rewind	Reverse	Load	Unload
0	Illegal request	x	x	x	x	x	x	x	x	x	x
1	Leader	x	x	x	x	x	x		x		
2	BOT missing							x		x	
3	Tape mark		x		x				x		
4	BOT/EOT hole		x	x	x	x			x		
6	Write protected	x	x	x	x	x	x	x	x	x	x
7	B Side	x	x	x	x	x	x	x	x	x	x
9	Rewind time out							x		x	
11	Sequence error	x	x	x	x	x	x	x	x	x	x
12	Incorrect length		x								
13	CRC error		x	x	x						
14	Throughput error		x	x	x						
15	Not operable	x	x	x	x	x	x	x	x	x	x

DRTP02 *Teller Terminal Printer* **DRTP02**

This status code is valid for teller terminal printers PTS6221, 6222 or 6223.
The following bits may be set by this driver:

Bit	Meaning	Test Status	Write	Position Voucher	Cut Journal Tape	Perforate	Grasp	Release Voucher
0	Illegal request	x	x	x	x	x	x	x
8	Recovery on request		x	x				
10	End of journal tape or Voucher out	x	x	x	x	x		x
13	Code check error		x	x				
15	Not operable	x	x	x	x	x	x	x

DRTP03 *Teller Terminal Printer* **DRTP03**

This status code is valid for teller terminal printer PTS6371.
The following bits may be set by this driver:

Bit	Meaning	Test Status	Write	Position Document	Open Document	Set Printer Parameters	Set Document Parameters	Release Document
0	Illegal request		x	x	x	x	x	
10	End of journal/document out	x	x	x				
13	Code check error		x					
15	Not operable	x	x	x	x	x		x

DRTW01 *Console Typewriter* **DRTW01**

This status code is valid for typewriter PTS6862. Alphanumeric characters in the range /20—/5F are sent from the user buffer to the printer. Codes /60—/7F are reduced by /20 giving /40—/5F.

The following bits may be set by this driver:

Bit	Meaning	Read	Write
0	Illegal request	x	x
9	Time out	x	
12	Incorrect length	x	
13	Code check error	x	x
14	Throughput error		

This status code is valid for files which are held on PTS6875 or 6876 diskdrives and flexible disks.

Bit	Meaning	Sequential Read	Sequential Write	Release Exclusive Access	Random Read	Random Write	Random Delete	CLOSE file	Indexed random read	Indexed rewrite	Indexed delete	Indexed insert	Indexed read next	Get current data record number	Get current index record number
0	Request error	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1	Key no found								x	x	x		x		
2	Record protected	x	x		x	x	x		x	x	x	x	x		
3	End of File	x			x	x	x		x	x	x	x	x		
4	No Data	x			x				x	x	x		x		
5															
6	Next key same								x		x	x	x		
7	Retries performed	x	x		x	x	x	x	x	x	x	x	x		
8	New Volume loaded	x	x		x	x	x	x	x	x	x	x	x		
9															
10	End of medium		x		x	x	x		x	x	x	x	x		
11															
12	Incorrect length	x			x				x				x		
13															
14	Disk I/O error	x	x		x	x	x	x	x	x	x	x	x		
15	Disk not operable	x	x		x	x	x	x	x	x	x	x	x		

Detailed information per access method. Only those bits are mentioned which need some more explanation.

GENERAL

Bit 0	Request error Set for request errors such as illegal order, unknown file code etc.
Bit 1	Key not found Set if the symbolic key required for indexed random instructions was not found in the index file.
Bit 2	Record protected Set if the accessed record is under "exclusive access" at the time of the read request, or the record is not under "exclusive access" and the record status indicates "USED" at the time of a write request.
Bit 3	End of file Set if the accessed record has a logical record number greater than the "last record number" (LRN) in the VTOC. In case of random read/write, the instruction is not aborted.
Bit 4	No data Set if the record status character indicates "free" at a read-request.
Bit 5	Not used
Bit 6	Next key same Set if the symbolic key in the next used index record is the same as in the current index record.
Bit 7	Retries performed The driver has retried an I/O action that was in error.
Bit 8	New volume loaded Set at the first request after a new volume has been loaded.
Bit 9	Not used
Bit 10	End of medium Set if the requested record is outside the physical space reserved for the file at creation time.
Bit 11	Not used
Bit 12	Incorrect length Set if the requested length is less than the record length at read request.
Bit 13	Not used
Bit 14	Disk I/O error Set for hardware errors, e.g. seek error, CRC-error, throughput error.
Bit 15	Disk not operable

ATTACH
DETACH

ATTACH
DETACH

The following bits may be set by this driver:

Bit	Meaning	Attach	Detach
0	Illegal request	x	x
9	Device not available	x	

APPENDIX C : CONTROL WORD INFORMATION

This information can be obtained with the CALL GETCW standard assembler subroutine.

Data management

The control word holds the logical record number.
It starts with 1 and is a 16 bit integer number.

Magnetic tape/Cassette

After completion of a read, the word contains the number of retries performed.

System Operator Panel

After a read the number of the activated switch is returned in this word. The switches are numbered from the right starting at 1.
It continues with 2, 3, 4, 5 etc.

Displays

The current cursor position is returned in the control word, which has the following lay-out :

0 7	8 . . . 15
line number	column number

Data Communication

The remaining time until time out is returned in the control word.

APPENDIX D : STANDARD ASSEMBLER SUBROUTINES

A number of Standard Assembler subroutines are held in the system library and may be called from CREDIT programs.

EMPTY *Empty Test* **EMPTY**

Syntax: [statement-identifier] ␣ CALL ␣ EMPTYT, data-item-identifier

Description: The data-item-referenced by data-item-identifier is tested for an empty value. Data item types binary, decimal and string are allowed.
The following values are considered as empty:
a) Binary-data-item, zero (0)
b) Decimal-data-item, all spaces (X'F')
c) String-data-item, all null characters (X'00')

Condition register: = 0 not empty
≠ 0 empty

Example: CALL EMPTYT, DEC2

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine.
Operand-1 is a reference to a binary, decimal or string data item.

GETCW *Get control word* **GETCW**

Syntax : [statement-identifier] ␣ CALL GETCW, data-set-identifier, data-item-identifier

Description : The control word of a data set indicated by data-set-identifier is stored in a binary data item indicated by data-item-identifier.

Condition register : unchanged

Example : CALL GETCW, DSDK, CONTRW

Intermediate code format :

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

Byte 1 contains the operation code (X'30')
Byte 2 is a reference to an external system routine
Operand-1 is a reference to a data set
Operand-2 is a reference to a binary data item

FMOVE *Format Move* **FMOVE**

Syntax: [statement-identifier] CALL FMOVE.
data-item-identifier, format-list-identifier

Description: The format-list referenced by format-list-identifier is copied into the string-data-item referenced by data-item-identifier. If the string-data-item is longer than the format list, remaining bytes are filled with the TEXT format-list-item.

Condition register: Unchanged.

Example: CALL FMOVE, STR1, FORM5

Intermediate object code:

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

Bytes 1 and 2 are filled by the system
operand-1 is a reference to a data set.
operand-2 is a reference to a format list.

ICLEAR *Clear data item* **ICLEAR**

Syntax: [statement-identifier] ␣ CALL ␣ ICLEAR, data-item-identifier

Description: The data item referenced by data-item-identifier will be cleared. Data-item types binary, decimal or string are allowed. Clearing of a data item will result in:

a) Binary-data-item is set to zero.
b) Decimal-data-item is set to all spaces (X'F')
c) String-data-item is set to null characters (X'00')

Condition register: Unchanged

Example: CALL ICLEAR, DEC5

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to an external system routine.
Operand-1 is a reference to a binary, decimal or string data item.

MASK *Mask function* **MASK**

Syntax : [statement-identifier] LI CALL MASK, data-item-identifier-1, data-item-identifier-2.

Description : From the two binary data items, indicated by data-item-identifier-1 and data-item-identifier-2, the logical product is taken and compared to zero. The result is stored in the condition register. This function is useful after a XSTAT instruction to examine the device dependent status.
The contents of the binary data items are not changed.

Condition register :
= 0 if result is zero
= 1 if result is positive
= 2 if result is negative

Condition mask :

0	1	2	3	4	5	6	7
= 0	> 0	< 0	—	≠ 0	⩽ 0	⩾ 0	Unconditional

Example : CALL MASK,MK1, STATUS

Intermediate code format :

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

Byte 1 contains the operation code (X'30')
Byte 2 is a reference to an external system routine
Operands-1,2 are references to binary data items

TYPET *Type Test* **TYPET**

Syntax: [statement-identifier] CALL TYPET, data-item-identifier-1, data-item-identifier-2

Description: Data-item-type of the data-item referenced by data-item-identifier-2 is tested and type is returned in the binary data-item referenced by data-item-identifier-1.

Following values are returned:

a) 1 for binary-data-item
b) 2 for decimal-data-item
c) 3 for string-data-item.

Condition register: Unchanged.

Example: CALL TYPET, BIN1, BIN2

Intermediate code format:

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

Bytes 1 and 2 are filled by the system
Byte 2 contains a reference to a system routine
Operand-1 is a reference to a binary data item
Operand-2 is a reference to a binary, decimal or string data item.

APPENDIX E : CHARACTER SET ISO-CODE

(recommendation R646)

Character/function declaration

Hex.	ISO Mnemonic	Declaration
0.0	NUL	Null
0.1	SOH	Start of Heading
0.2	STX	Start of Text
0.3	ETX	End of Text
0.4	EOT	End of Transmission
0.5	ENQ	ENQUIRY
0.6	ACK	Acknowledge
0.7	BEL	Bell
0.8	BS	Backspace
0.9	HT	Horizontal Tab
0.A	LF	Line Feed
0.B	VT	Vertical Tab
0.C	FF	From Feed
0.D	CR	Carriage Return
0.E	SO	Shift Out
0.F	SI	Shift In
1.0	DLE	Data Link Escape
1.1	DC1	X-On Reader
1.2	DC2	X-On Puncher
1.3	DC3	X-Off Reader
1.4	DC4	X-Off Punch
1.5	NAK	Negative Acknowledge
1.6	SYN	Synchronous Idle
1.7	ETB	End of Transmission Block
1.8	CAN	Cancel
1.9	EM	End of Medium
1.A	SUB	Substitute
1.B	ESC	Escape
1.C	FS	File Separator
1.D	GS	Group Separator
1.E	RS	Record Separator
1.F	US	Unit Separator
2.0	SP	Space
7.F	DEL	Delete/Rub Out

There are 12 positions available for national usage marked with 

Per I/O device, the character set may vary.

$b_7b_6b_5$ →	000	001	010	011	100	101	110	111
$b_4b_3b_2b_1$ ↓ col / row	0	1	2	3	4	5	6	7
0000 0	NUL	DLE	SP	0	@	P		p
0001 1	SOH	DC1	!	1	A	Q	a	q
0010 2	STX	DC2	"	2	B	R	b	r
0011 3	ETX	DC3	#	3	C	S	c	s
0100 4	EOT	DC4	$	4	D	T	d	t
0101 5	ENQ	NAK	%	5	E	U	e	u
0110 6	ACK	SYN	&	6	F	V	f	v
0111 7	BEL	ETB	'	7	G	W	g	w
1000 8	BS	CAN	(	8	H	X	h	x
1001 9	HT	EM	)	9	I	Y	i	y
1010 A	LF	SUB	*	:	J	Z	j	z
1011 B	VT	ESC	+	;	K		k	
1100 C	FF	FS	,	<	L		l	
1101 D	CR	GS	-	=	M		m	
1110 E	SO	RS	.	>	N		n	
1111 F	SI	US	/	?	O		o	DEL

APPENDIX F : SCREEN MANAGEMENT

This appendix gives a description of screen management as it may be called by an application for simplifying display handling. The module is written in CREDIT and supplied on the system disk, with module name SCREEN.

F.1 *Introduction*

This module can handle different types of applications which are used for e.g. data entry, inquiry, register updating, transactions etc. It uses a display, print device and keyboard for communication.

Entering the module is done with a Perform instruction followed by the procedure name. When control is passed to the module it will display the required screen layout as defined in a format list, which is prepared outside the package. Also input fields and corresponding data items are fully described in this format list.

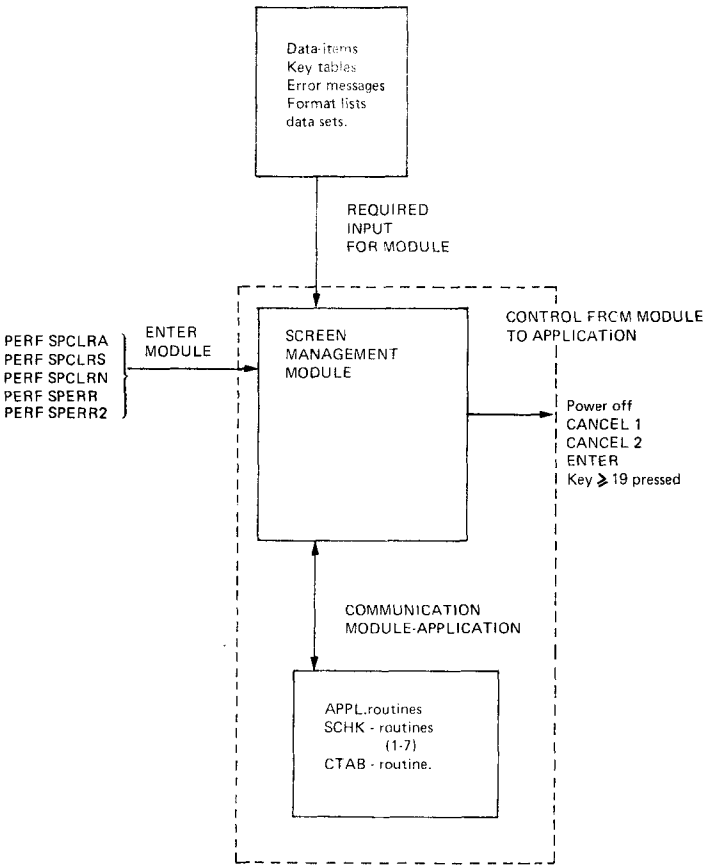
Tabulation functions, error messages, checks on input fields (APPL, MAXL etc.), function keys to terminate input, editing mode allowing cursor positioning, deletion or insertion of a character in a current input field and hardcopy facilities are all included in the module. When the transaction is terminated or e.g. a CANCEL key is pressed control will be given to the application.

Data items, keytables, datasets, error messages, format lists (for screen layout) and routines which are executed on a APPL,CTAB or SCHK option in the current input field, are defined outside the module, using standard names.

The VERIF option is not yet included in the module but the user has the possibility to change the module because all is written in the CREDIT language.

Only input fields of the type FKI are handled. FINP-fields have to be programmed outside the package. When the ENTER-key (end of transaction) is pressed, screen management will check if compulsary fields are filled. If not, the cursor will be positioned in such a field and data must be entered before the transaction can be completed.

It is advised to read also chapter 1.4.3.1 in this manual, which also describes the buffer handling.


Data-items
Key tables
Error messages
Format lists
data sets.
REQUIRED
INPUT
FOR MODULE
PERF SPCLRA
PERF SPCLRS
PERF SPCLRN
PERF SPERR
PERF SPERR2
ENTER
MODULE
SCREEN
MANAGEMENT
MODULE
CONTROL FROM MODULE
TO APPLICATION
Power off
CANCEL 1
CANCEL 2
ENTER
Key ≥ 19 pressed
COMMUNICATION
MODULE-APPLICATION
APPL.routines
SCHK - routines
(1-7)
CTAB - routine.

F.2 Using Screen

Before the module is [illegible], describing the screen layout and giving the options of input fields, must be [illegible] the Attach Format instruction (ATTFMT) has to be executed. When [illegible] the prompts must be displayed a boolean data item SPPROMPT must be set [illegible] has to be zero and prompts will not be displayed.

Entering the module [illegible] execution of one of the following perform instructions:

[illegible]
PERF SPCLRS
[illegible]
[illegible]

SPCLRA: When [illegible] then following actions are performed on calling [illegible]

- [illegible] input fields.
- Display [illegible]
- Cursor [illegible] location of current FKI-field. If there is no current [illegible] is made current.
- Read [illegible]
  The first [illegible] is matched with keytable SPKTAB1, following characters are checked with [illegible]
- Check [illegible] otherwise display error message
- Check is performed [illegible] the input field if APPL, CTAB or SCHK option [illegible] specified.
- Input fields [illegible] moved to corresponding data item
- Set boolean [illegible] to indicate that a data item contents is changed/updated
  (This data [illegible] has to be reset outside the module).
- Make next [illegible] current, when End of Item key is pressed.
- When [illegible] key is pressed, control will be passed to the application. Binary [illegible] is set to value 3.

When SPPR[illegible] = 0

- Clear [illegible] the screen and corresponding data items.
- Cursor [illegible] character position of current FKI-field. If there [illegible] current [illegible] field is made current.

Following [illegible] SPPROMPT = 1

SPCLRS: When [illegible] SPPROMPT = 1, then following actions are performed on [illegible]

- Clear [illegible] input fields, except those which have option NCLR set [illegible]
- [illegible] which are not not cleared. (NCLR set).
- Cursor [illegible] location of current FKI-field. If there is no current [illegible] current.

For [illegible] SPPROMPT = 1. This routine allows displaying [illegible] data item, when in the corresponding FKI-input [illegible]

When [illegible]

- [illegible] corresponding data items, except when [illegible] NCLR [illegible] fields.
- Cursor [illegible] current FKI-field. If there is no [illegible]

Following items the same as when SPPROMPT = 1

SPCLRN: When the boolean data item SPPROMPT = 1, then following actions are performed on calling the module with this name;

- No clearing of input fields and corresponding data items is performed.
- Display entire format and input fields.
- Cursor is located at first character position of current FKI-field. If there is no current field, the first input field is made current.

Following items are same as SPCLRA with SPPROMPT = 1. This routine makes it possible to recall a complete screen layout with its corresponding input field information.

When SPPROMPT = 0

- No clearing of input fields and corresponding data items is performed.
- Cursor is located at first character position of current FKI-field. If there is no current field, the first input field is made current.

Further the same as with SPPROMPT = 1

This routine makes it possible to enter the module and continue on a current field without any changing in the format on the screen.

SPERR: When an error is detected outside the module in an application routine other than SPCHKx, SPAPPL or SPTCHK, the user may call the module with this error routine name and pass the index of the error message in a binary data item SPBINW4. Screen management will handle this error as if it was detected in the package itself. A bell signal is sent and an error message will be displayed on the last line of the screen. (See also Error handling 5.1).

SPERR2: When an error is detected outside the module in an application routine other than SPCHKx, SPAPPL or SPTCHK, the user may call the module with this error routine name and pass the index of the error message in a binary data item SPBINW4. Screen management will examine and modify the current input field, and control is returned to the calling routine.
A bell signal is sent and an error message displayed on the bottom line of the screen (see also Error handling 5.1).

F.3 *Communication between Screen Management module and application*

Screen management checks the input field on options APPL (application), SCHK (check) and CTAB (conditional tabulation). Verification is not yet implemented in the package but has to be included by the user if required. When one of these options is specified, a subroutine outside the package is called with respectively the names SPAPPL, SPCHK1 through SPCHK7 or SPTCHK.

These routines are mentioned as externals (EXT) in the screen management module and require a corresponding entry (ENTRY) in the application module(s).

F.3.1 *SPAPPL. (application)*

After data is entered and the input is terminated screen management will check if the APPL option was specified for this input field. The interface between screen management and the routine SPAPPL is as follows:

Input to routine SPAPPL;

- SPINPUT, a string data containing the data input via the keyboard.
- SPBINW1, a binary data item containing the number of transferred characters on input.
- SPBINW3, a binary data item containing the APPL value as mentioned in the FKI input field definition. This value will be processed in the sub-routine.
- SPBINW2, a binary data item containing the converted end-if-item key ind .x in the key table.

Output from SPAPPL;

The application has to return in the binary data item SPBINW3 a value 0, 1, 2 or 3 which will result in the following actions in screen management.

(SPBINW3) = 0 The contents of data item SPINPUT will be moved to the data item of the current input field and displayed on the screen when the "REWRT" option is defined for this input field. Screen management continues according to the converted end-of-item key index as present in SPBINW2.

(SPBINW3) = 1 The contents of data item SPINPUT will be moved to the data item of the current input field and is always displayed on the screen. Screen management continues according to the converted end-of-item key index as present in SPBINW2.

(SPBINW3) = 2 The contents of data item SPINPUT is not moved to the data item of the current input-field. Cursor is set at the begin of the current input field and input on this field can be performed.

(SPBINW3) = 3 Error detected. Binary data item SPBINW4 contains an index for the error print out format. When the contents is zero, no error message is printed.
(See also error handling 5.1).

F.3.2 *SPTCHK (conditional tabulation)*

When a tabulation function key is pressed as forward, backward, upward etc., screen management searches for the required field.

When this input field is found and it has the CTAB option specified, screen management will call the routine SPTCHK, before setting the cursor in this field. The application routine SPTCHK has to check if this new current input field is wanted, or has to be continued on tabulating forward or backward to following/preceding fields. Once the tabulation direction, forward or backward, is set it will continue in this direction until the cursor is positioned. The cursor remains in its original position as long as CTAB option is specified for these input fields. The routine SPTCHK has to indicate in the binary data item SPBINW3 which action has to be taken by screen management.

Output from SPTCHK:

(SPBINW3)= 0 Tabulation is at correct input field. The cursor will be positioned at the first character position in this current field.

(SPBINW3) = 2 Tabulation must continue forward or backward. The tabulation will continue backward, by screen management if the initial (present in SPBINW2) tabulation key pressed was:

- tabulation backward
- tabulation right most
- tabulation upwards

All other keys pressed will result in continuing forward tabulation.

F.3.3 *SPCHK1 – SPCHK7*

After data is entered and input is terminated, screen management will check if the SCHK option was specified for this input field. Depending on the value (1 to 7) defined after SCHK, the routine SPCHK1, SPCHK3, SPCHK4, SPCHK5, SPCHK6 or SPCHK7 will be called. The user can perform a check on this input field.

Input to routine SPCHKx:

- SPINPUT, a string data item containing the data input via the keyboard.
- SPBINW1, a binary data item containing the number of transferred characters on input.
- SPBINW2, a binary data item containing the converted end-of-item key index in the key table.
- SPBINW4, a binary data item containing the value defined in SCHK option.

Output from SPCHKx:

This interface is the same as for the SPAPPL routine, also in SPBINW3 a value 0, 1, 2 or 3 has to be returned, with same significance.

Note: When both options SCHK and APPL are specified for the current input field, first the SPSCHKx routine will be performed and then the SPAPPL routine.

F.4. *Key tables used for Screen Management*

F.4.1 *General*

Screen Management uses three keytables to terminate input via a keyboard. These key tables have a standard layout for the function keys. The corresponding key codes have to be defined by the user in a module SPLITT.

When a certain function key is not required by the user, it has to be set to the keycode value x'FF' (e.g. CANCEL key 2, CANC2 EQUAL x'FF').

The keys in keytable SPKTAB1, are function keys which are allowed to be entered as input field. The keys in key table SPKTAB2, are function keys which are allowed to be entered at any other position of the current input field. The keys in key table SPKTAB3, are function keys which are only effective when the package is switched to edit mode.

Reading of input characters is performed as a keyboard input with echoing on the display. The function keys are not echoed on the display.

The first character read is checked if it is defined in SPKTAB1. When not present in the keytable and it is a numeric/alphanumeric character, the remaining part of the field on the display will be cleared, the input character is echoed on the display and the remaining character positions are displayed as a number of dots, corresponding to the maximum length of the current input field.

Remaining characters are input and checked with key table SPKTAB2.

If an illegal key code is entered, the acoustic alarm will sound and reading is continued.

After completion of the input, the remaining dots in the field are cleared and a check is performed if options as APPL and SCHK are defined for this input field, before continuing according to the pressed function key.

F.4.2 *Function keys in SPKTAB1*

This key table is used when the cursor is positioned at the first character position of the current input-field. It is also used after an error message is displayed, then only key codes at position 2, 3, 5 and 6 are relevant.

Position number in SPKTAB1	Significance
1	BACKSPACE. When this key is pressed, the cursor is moved one step to the left and the corresponding position on the display is replaced by a dot. If backspace is performed at the very first position of the current input field, the old contents of the data item belonging to this input field will be displayed.
2	CLEAR. The input field and corresponding data item are cleared, when the MAXL value is greater than zero. The cursor remains in the first position of the current input field.
3	[illegible]. The input field is displayed with the old contents of the belonging data item. The cursor remains at the first position of the current input field.

Position number in SPKTAB1	Significance
4	EOI. Common end of item key. Tabulation forward to the next input field is performed.
5	CANCEL1. A return to the application is performed with in the binary data item SPBINW2, the value 1. No checks are performed on the input field.
6	CANCEL2. Same as CANCEL1 but in SPBINW2, the value 2 is returned.
7	TFWD. Tabulation forward to the next input field. No action is taken if this input field is not present. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of this compulsory field. The compulsory field becomes current.
8	TBWD. Tabulation backward to the previous input field. No action is taken if this input field is not present. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of this compulsory field. The compulsory field becomes current.
9	THOME. Tabulation to the first input field of this format list.
10	TLDOWN. Tabulation to the first input field on the next line. No action is taken if this input field is not present. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of this compulsory field. The compulsory field becomes current.
11	TLEFT. Tabulation to the most left input field on the current line. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of this compulsory field. The compulsory field becomes current.
12	TRIGHT. Tabulation to the most right input field on the current line. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of the compulsory field. The compulsory field becomes current.
13	TDOWN. Tabulation to the input field on the next line, with a starting column nearest to the starting column of the current input field. When two nearest columns are found, tabulation will be to the left input field. When no input field is found, following lines will be searched. No action is taken if the input field is not present. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of the compulsory field. The compulsory field becomes current.

Position number in SPKTAB1	Significance
14	TUP. Tabulation to the input field on the preceding line, with a starting column nearest to the starting column of the current input field. When two nearest columns are found, tabulation will be to the left input field. When no input field is found, preceding lines will be searched. No action is taken if the input field is not present. If an empty compulsory field is found in an earlier defined input field, the cursor is positioned at the begin of the compulsory field. The compulsory field becomes current.
15	COPY. A hardcopy of the entire screen is made on the print device.
16	DUPL. Move of the contents of the duplication data item, as defined by the DUPL option in the current input field, to the field SPINPUT and display. When no DUPL option defined, an error message will be displayed.
17	EDIT. Set to edit mode. (See SPKTAB3 description 4.4).
18	ENTER. Ends the handling of this complete format if all compulsory fields are filled and returns to the application. When not all compulsory fields are filled, an error message is displayed and the cursor is positioned on the first not filled compulsory field, and this field becomes current.
19	Application functions keys. When a function key with a position number 19 or greater in the key table is pressed, a return will be performed to the application. In data item SPBINW2 is returned the value 4 for a key in position 19. For a key in position 20 the value 5 is returned etc.

F.4.3 *Function keys in SPKTAB2*

The key table has the same layout as key table SPKTAB1 and is used when the cursor is positioned anywhere in this input field except on the first position. When a function key is required only in SPKTAB1 and not in SPKTAB2, then the corresponding key position in SPKTAB2, has to get a key code X'FF'.

Keys from position 7 and up will first be handled as an end-of-item key and after the input data is found correct a check on SCHK and APPL options is done. Then the function required is executed.

For position numbers in key table and significance. See SPKTAB1.

F.4.4 *Functions keys in SPKTAB3, edit mode*

The edit mode is entered by pressing the function key for edit mode, which key code is defined in key table SPKTAB1 and SPKTAB2. Furthermore the MAXL option must be greater than zero.

A change to edit mode may be done during data input for the current field e.g. three character positions after a wrong character was entered and this has to be changed before continuing.

In this mode more functions are available to update on character level in the current input field as e.g. non destructive space, delete a character, insert a character etc.

After input is terminated and checked the data item belonging to this current input field will be updated.

If an illegal character is entered the acoustic alarm will sound and editing continues. The character at the cursor position will be overwritten by numeric or alphanumeric keys.

Position number in SPKTAB3	Significance
1	→NON-destructive space. Moves the cursor one step to the right. Accoustic alarm sounds if trying to exceed the effective item limits.
2	← NON-destructive backspace. Moves the cursor one step to the left. No action if most left position is reached.
3	INS. Insert a character at current cursor position. The character under the cursor and the characters right of the cursor in the current field are shifted one step to the right with truncation. One space character will be inserted at the cursor position and the cursor is not moved.
4	DEL. Delete character at current cursor position. The other characters to the right, in the current field, are shifted one step to the left.
5	CLEAR1. The input field and corresponding data item are cleared. Cursor is positioned at the first character position of the field. Out of edit mode.
6	CLEAR2. The input field is displayed with the old contents of the data item belonging to this field. Out of edit mode.
7	CLEAR3. Clear field from current cursor position up to the most right position. Out of edit mode.
8	The same keys as for SPKTAB1 and SPKTAB2 as mentioned from position 4 and up. Out of edit mode.

F.5. *Error handling*

F.5.1 *Errors detected in the Screen Management module*

The module checks:

- the number of characters entered is less than specified in MINL.
- an I/Ø error during input.
- Illegal end-of-item key pressed, e.g. duplication but in the input field the DUPL option is not specified.
- Compulsory field(s) not filled when the ENTER key is pressed.

Before the error message is displayed on the last line on the screen, the acoustic alarm sounds. The error messages are to be defined by the user in a table SPFTBERR and an index for these messages is present in binary data item SPBINW4.
When this index is zero no error message is displayed and reading is resumed in the current input field. After the error message is displayed the cursor will be placed at the current input field. Input is done via the keyboard with using key table SPKTAB1. Only the functions CLEAR1, CLEAR2, CANCEL1, CANCEL2, and EDIT are relevant for this input, and clearing of the current input field is done if the input length as present in SPBINW1 is not zero.

If SPBINW1 is containing zero no clearing of the current input field is done. The error message is cleared and there will be continued according to keytable SPKTAB1.

Contents of SPBINW4 (error message index)	Significance
0	No error message displayed.
1	Number of characters input is less than stated in the MINL option.
2	Not used.
3	I/O-error (e.g. time out, throughput error etc.).
4	Illegal end-of-item key pressed. (e.g. duplication wanted but option not specified in the input field).
5	Compulsory field not filled after pressing the ENTER key.
6	This index value and up may be used by the routines SPAPPL and checkroutines SPCHK1 - SPCHK7.

F.5.2. *Errors detected outside the Screen Management module*

When an error is detected outside the package, it is possible to enter the module in the error handling section by calling the routine "SPERR" (PERF SPERR) or "SPERR2" (PERF SPERR2).

The binary data item SPBINW4 should contain the error message index and SPBINW1 must be unequal to zero if clearing is wanted, or zero if clearing is not wanted.

When the package is entered via SPERR, error handling is the same as described in 5.1. By pressing the CLEAR1, CLEAR2 key or the CANCEL1, CANCEL2 key, the user may keep control of the screen management module or leave the module.

When the package is entered via SPERR2, only the current input field can be modified. All keys, except the EDIT key, are considered as ENTER keys, though a hardcopy is obtained when the COPY key is pressed.

The routines SPCHKx and SPAPPL are assumed to be part of the screen management module. (See 3.1 and 3.3).

F.6. *Control from package to application*

Control is passed to the application when:

- power off is detected
- CANCEL1 key is pressed.
- CANCEL2 key is pressed.
- ENTER key is pressed.
- An application defined, function key is pressed. ($\geqslant$19)

This information is passed to the application in binary data item SPBINW2. Also the boolean data-item SPCHANGE is set to indicate that at least one of the data items belonging to the input field is changed. It should be reset outside the module.

In the application has to be decided what to do with the result in SPBINW2.

Contents of SPBINW2	Significance
0	Power off detected. Return to application. Screen may be cleared.
1	CANCEL1 key pressed. Return to application. No check on input performed. Current input field (FKI-type) and corresponding data item is cleared.
2	CANCEL2 key pressed. Return to application. No check on input performed. Current input field (FKI-type) and corresponding data item is cleared.
3	ENTER key pressed. Return to application. No empty compulsory field(s) are found.
4 and up	Application defined function key is pressed. For SPKTAB1, a key at position 19 and up. For SPKTAB2, a key at position 19 and up. For SPKTAB3, a key at position 23 and up. Key position 19 (SPKTAB1, SPKTAB2) or 23 (SPKTAB3), will be converted to index value 4 in SPBINW2. Position number 20, respectively 24 will be converted to index value 5 etc.

F.7. *Required definitions outside Screen Management module*

Screen Management uses data items which are defined in the data division of the application. In the module is referred to the data division by means of a DDUM SPDDIV directive. When the user wants to use for his data division an other name, then the name SPDDIV in the module has to be updated. The equates for the key tables and the format lists describing the error messages, are expected to be defined in a module SPLITT. The package uses the instruction INCLUDE SPLITT, LIST. To find the names used in key tables SPKTAB1, SPKTAB2 and SPKTAB3, the module SCREEN must be translated. Error messages are expected to be defined in the format table SPFTBERR. The format list of each error message must start with the FSL directive.

Format list describing the screen layout are defined in the application and do not have to be included in module SPLITT.

Data sets to be defined in the data division and used by screen management.

Name	Significance
SPDSPRT	Data set with fixed buffer for hard copy device. The buffer size must have at least the same size as the one used for the display. (Buffers may be shared).
SPDSSCRN	Data set with fixed buffer for the display. Buffer size must be large enough to hold the maximum number of characters (e.g. one line) inclusive the control characters. (Buffers may be shared).
SPDSDYKB	Data set for the keyboard.

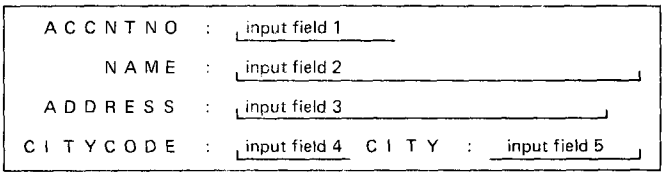
A format control I/O declaration (FMTCTL) has to be defined for input dataset and output dataset, which are used in the format control I/O instructions. (e.g. DYKI, DISPLAY).

Data items to be defined in the data division and used by screen management.

Name	Significance
SPBINW1	A binary data item used to contain the number of characters transferred during input or used as work item.
SPBINW2	A binary data item used to contain the (converted) end-of-item key.
SPBINW3	A binary data item used to contain the APPL-value.
SPBINW4	A binary data item used to contain the SCHK value (1 up to 7) or error code index.
SPCHANGE	A boolean data item which is set by screen management to indicate that a data item belonging to a current input field is changed. Resetting has to be done outside the module.
SPPROMPT	A boolean data item used to indicate that the prompt texts should be displayed. If set (TRUE), the entire format is displayed including the prompts. If reset (FALSE), only the data items belonging to the input fields are displayed and not the prompts. SPPROMPT has to be set or reset outside the module.
SPINPUT	A string data item used as keyboard input buffer. The size must be large enough to contain the maximum input length, inclusive the end-of-item key.
SPERCALL	A boolean data item used by screen management to indicate whether the SPERR or SPERR2 entry is used. This data item is set to one when SPERR2 is called.
SPSTRGW1	A string data item (≥ 2 characters) used by the error routine.

F.8. *Example of a coded format*

The following picture is defined:



field 1 :
- numeric
- compulsary field
- exactly 10 digits long
- no end-of-item required
- formatting after input wanted
- CDV-10 check should be performed

field 2 :
- alphanumeric
- 2 - 35 characters long

field 3 :
- alphanumeric
- 2 - 30 characters long
- duplication from item ITEM10 in an other picture is allowed

field 4 :
- numeric
- exactly 5 digits long
- no end-of-item key required
- formatting after input wanted
- after entering the field, the input is used as key in a file for corresponding CITY.
  When found, the CITY is displayed in the next field (field 5). If not found the cursor will be placed at the next field.

field 5 :
- alphanumeric
- 2 - 20 characters long

The corresponding format can look as follows:

```
PICT1    FRMT
         FSL
         FCOPY      =  ' A C C N T N O : '
         FKI        11, ME, MINL=10, MAXL=10, NEOI, REWRT,
                    SCHK=1
         FMEL       'XXXXXXE-XXXX', ITEM1
         FNL
         FCOPY      = '   N A M   E : '
         FKI        11, ALPHA, MINL=2, MAXL=35
         FCOPY      ITEM2
         FNL
         FCOPY      = ' A D D R E S S : '
         FKI        11, ALPHA, MINL=2, MAXL=30, DUPL=ITEM10
         FCOPY      ITEM3
         FNL
         FCOPY      = ' C I T Y C O D E : '
```

```
FKI       11, MINL=5, MAXL=5, NEOI, REWRT, APPL=1
FMEL      'XXXBXX', ITEM4
FCOPY     = ' C I T Y : '
FKI       23, ALPHA, MINL=2, MAXL=20
FCOPY     ITEM5
FMEND
```

APPENDIX G : STANDARD CREDIT SUBROUTINES

A number of CREDIT subroutines are held in the system library and may be called by a CREDIT program.

STRINP *String input* **STRINP**

Syntax: [statement-identifier] CREDIT STRINP, data-item-identifier-1, data-item-identifier-2, data-item-identifier-3, data-item-identifier-4, index-identifier-1, index-identifier-2, data-item-identifier-5.

Type: CREDIT subroutine call.

Description

Before calling the subroutine, the requested picture description (format-list) has to be made current with the ATTFMT instruction.
Character fields contained in a string-data-item are moved to data-items, belonging to FKI-and/or FINP fields in the current format list. (The MOVE conversion rules are applied). The character fields, in the input string-data-item, are separated by input separation characters defined by the user. These characters are in the range from X'00' to X'FF'.
The character fields are moved to consecutive data-items in the format-list. The start position in the input string is indicated with character position number (first character position is zero).
Field sequence numbering is selected by string field type in a binary-data-item. Two indexes hold the first and last field number at which consecutive copying is started and has to be stopped.
When an error is detected an error code will be returned.

Data-item-identifier-1, is a binary-data-item holding the field type.
0 = FKI
1 = FINP
2 = FKI/FINP
This data-item is not changed by the sub-routines.

Data-item-identifier-2, is a string-data-item containing the input character fields.
This data-item is not changed by the sub-routine

Data-item-identifier-3, is a binary-data-item which holds the start position of the input string referenced by data-item-identifier-2.
First character position is zero.
This data-item will point to the next field to be moved.

Data-item-identifier-4, is a string-data-item holding the input separation character in the first position.
This data-item is not changed by the sub-routine.

Index-identifier-1, is a binary-data-item containing the first field number. (May not be zero).
This data-item will point to the last moved field.

Index-identifier-2, is a binary-data-item containing the last field number. (May not be zero).
This data item is not changed by the sub-routine.

Data-item-identifier-5, is a binary-data-item in which a code is returned.
0 = OK, all fields are moved as required.
3 = Not OK, End of format-list is reached or end of input string is reached.

STRINP *Continued* **STRINP**

Example:

```
PERF    STRINP, TYPE, INPSTR, STARTO, SEPCHAR,
INDX1, INDX2, RETCODE
```

STROUT *String Output* **STROUT**

Syntax: [statement-identifier] ␣ PERF ␣ STROUT, data-item-identifier-1, data-item-identifier-2, data-item-identifier-3, data-item-identifier-4, index-identifier-1, index-identifier-2, data-item-identifier-5.

Type: CREDIT subroutine call

Description: Before calling this subroutine, the requested picture description (format-list) has to be made current with the ATTFMT instruction. The contents of data-items belonging to FKI- and/or FINP fields in the current format-list are moved in consecutive order to an output string-data-item. After moving a data-item contents to the output string, a unit separation character will be inserted, before the next data-item, in sequence, is moved.
Decimal-data-items will be converted to ISO-7 representation and leading blanks are skipped. An empty item results in only a unit separation character.
These separation characters may range from X'00' to X'FF'.
The start position in the output string is indicated with character position number (First character position is zero).
Field sequence numbering is selected by storing field-type in a binary-data-item.
Two indexes hold the first and last field number at which consecutive copying to the output string is started and has to be stoppped.
When an error is detected an errorcode is returned.

Data-item-identifier-1, is a binary-data-item holding the field-type
0 = FKI
1 = FINP
2 = FKI/FINP
This data-item is not changed by the sub-routine.

Data-item-identifier-2, is a string-data-item in which the contents of the data-items of the format list, will be packed.

Data-item-identifier-3, is a binary-data-item which holds the start position of the output string referenced by data-item-identifier-2.
First character position is zero.
This data-item will point to the next free area.

Data-item-identifier-4, is a string-data-item holding the unit separation character in the first position.
This data-item is not changed by the sub-routine.

Index-identifier-1, is a binary-data-item containing the first field number. (May not be zero).
This data-item will point to the last moved field.

Index-identifier-2, is a binary-data-item containing the last field number. (May not be zero).
This data-item is not changed by the subroutine.

Data-item-identifier-5, is a binary-data-item in which a code is returned.
0 = OK, all fields are moved as requested
3 = not OK, End of format list is reached or end of output string is reached.

STROUT

Continued

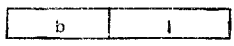
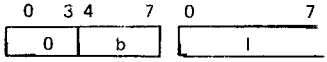
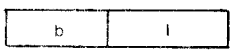
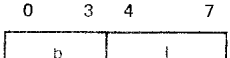
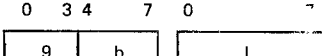
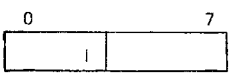
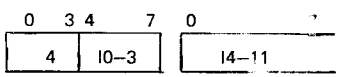
STROUT

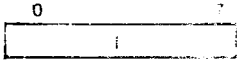
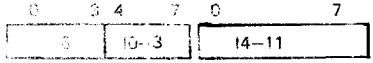
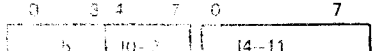
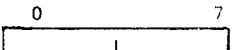
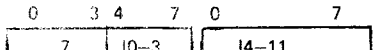
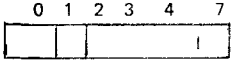
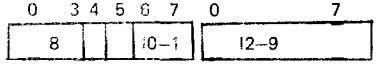
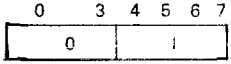
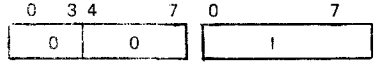
Example:

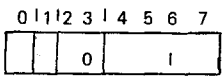
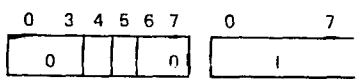
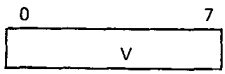
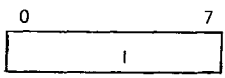
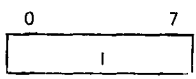
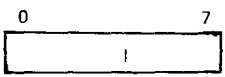
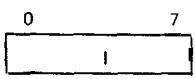
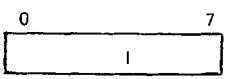
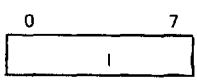
```
PERF   STROUT, TYPE, OUTSTR, START0, SEPCHAR,
INDX1, INDX2, RETCODE
```

APPENDIX H : OBJECT CODE FORMAT

When ADRMOD = 2 is specified in the OPTNS directive, the data-item, data-set, formal parameter, literal, picture, keytable and format references are extended with one byte. The layout of the object code is then different from the one-byte representation.

<table>
<tr><th>REFERENCE TYPE</th><th>OBJECT CODE FORMAT
ADRMOD = 1</th><th>ADRMOD = 2</th></tr>
<tr><td>Data-Item or ARRAY</td><td> 
</td><td> 
</td></tr>
<tr><td>ARRAY used in PERF/PLIST parameter list</td><td> 
</td><td> 
</td></tr>
<tr><td>Boolean data item</td><td> 
</td><td> 
</td></tr>
<tr><td colspan="3">b is the workblock index, ranging from X'1' to X'F'
I is the data item index, ranging from X'0' to X'F'
(ADRMOD = 1) or from X'00' to X'FF' (ADRMOD = 2).
For boolean data items I is in the range from X'0' to X'F' in both addressing modes.</td></tr>
<tr><td>LITERAL constant</td><td> 
</td><td> 
</td></tr>
<tr><td colspan="3">I is the literal index, ranging from X'10' to X'FF' (ADRMOD = 1) or from X'100' to X'FFF' (ADRMOD = 2).</td></tr>
</table>

REFERENCE TYPE	OBJECT CODE FORMAT ADRMOD = 1	ADRMOD = 2
Key table		 I is the key table index, ranging from X'10' to X'FF' (ADRMOD = 1) or from X'100' to X'FFF' (ADRMOD = 2).
Picture		 I is the picture index, ranging from X'10' to X'FF' (ADRMOD = 1) or from X'100' to X'FFF' (ADRMOD = 2).
Format list		 I is the format list index, ranging from X'10' to X'FF' (ADRMOD = 1) or from X'100' to X'FFF' (ADRMOD = 2).
Data set		 I is the data set index, ranging from X'10' to X'3F' (ADRMOD = 1) or from X'10' to X'3FF' (ADRMOD = 2). Bits 0/1 or 4/5 are used for transfer of the wait bit and echo/exclusive access bit.
Formal parameter (except for data sets)		 I is the position number of the formal parameter ranging from X'0' to X'7' for both addressing modes

REFERENCE TYPE	OBJECT CODE FORMAT	
	ADRMOD=1	ADRMOD=2
Formal parameter (data sets)		
	I is the position number of the formal parameter, ranging from X'0' to X'7' for both addressing modes. Bits 0/1 or 4/5 are used for transfer of the wait and echo/exclusive access bits.	
Immediate values (e.g. list length)		
	V is a value ranging from X'00' to X'FF' (ADRMOD=1)	
Subroutine in PERF		
Subroutine in CALL		
Branch in branch instructions		
	I is the index into the proper address table for performs, calls or branches.	
:FMTITEM	X'0C'	X'000C'