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:

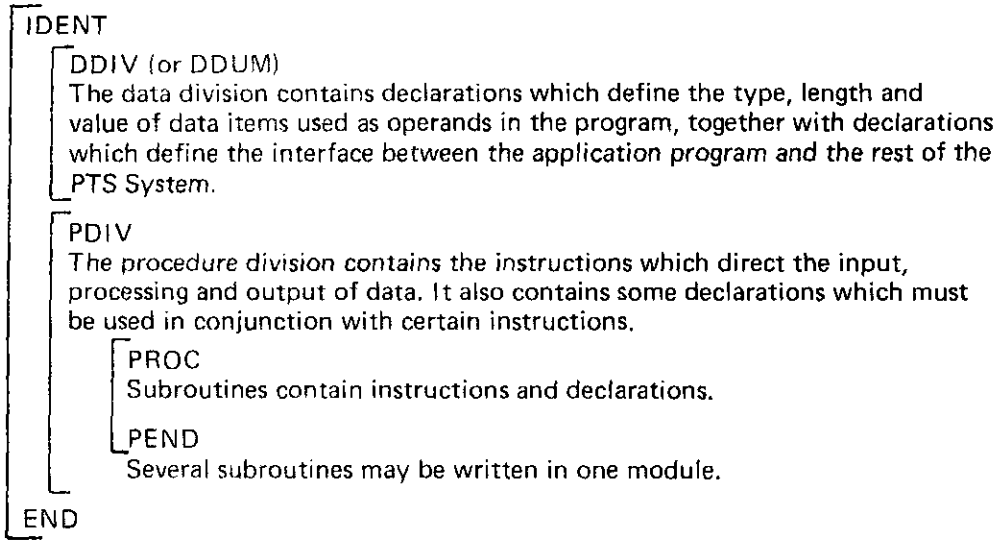
:: =	is defined as
␣	space
[ ]	the syntatic items between these square brackets may be omitted
{ }	select one of the items between the braces
a|b	select either a or b. This has the same meaning as braces. It is used with long strings.
...	ellipsis indicates that the last syntatic item may be repeated.

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.