NATURAL Essentials

a self-study programming course by

Stephen Paul Simpson

(page intentionally blank)

Disclaimer

This material is not intended as a master reference for NATURAL and should not be used as such. Although the author has made every effort to be accurate, the accuracy and completeness of the information presented is not guaranteed. The author accepts no responsibility for loss of any kind resulting in any way from use of this material. Except as a former employee, the author is not connected with Software AG.

Copyright & License Information

This original work is the intellectual property of Stephen Paul Simpson. Registered Users are welcome to download and print this material for personal, non-profit use. To become a Registered User, please visit www.spsimpson.com/natural/course/download/. Although originally conceived as tool for individuals, some Software AG clients have chosen this course as a primer for new employees and contractors. Your organization can do the same by purchasing a modestly priced license to make and distribute an unlimited number of printed copies. The same license also allows you to install the course on a local area network. For more information about this option, please visit www.spsimpson.com/license.

Copying or installing on a network without a license, altering the text, distributing/publishing individual parts of the course, distributing/publishing outside your organization and commercial/for-profit use all constitute infringements of copyright and will be prosecuted to the fullest extent of the law.

Update History

Version 1.00 published January 9, 1999.
Version 2.00 published August 24, 1999.
Version 2.10 published January 22, 2000.

Since I actively refine and improve NATURAL Essentials™, what you're reading may not be the current version. Please visit www.spsimpson.com for the latest version.

Trademarks

"NATURAL", "ADABAS" and "CONSTRUCT" are trademarks of Software AG. "Acrobat" and "Acrobat Reader" are trademarks of Adobe Systems, Inc. "NATURAL Essentials", "Only NATURAL!", "NATURAL Instructor Kit" and "NAT-U" are trademarks of Stephen Paul Simpson. Any other trademarks used are the property of their respective owners.

Contents

Introduction

About the course

This course includes all the information a technical person competent in another programming language needs to begin using NATURAL effectively and efficiently. For those students who need the interaction of formal classroom training, this material is intended to be a useful complement - included are many techniques and personal insights they don't always teach you in class.

It is presented in 3 parts. Part 1 covers concepts, part 2 covers syntax, and part 3 contains complete source listings and narratives for a simple application to demonstrate how things fit together. There are 3 appendices. Appendix A covers the essential tools and commands to build NATURAL applications. Appendix B contains naming convention guidelines. Appendix C contains a quiz to test your knowledge of NATURAL. At the end, please take a few minutes to give me your feedback - your input will make this course better.

Since I actively refine and improve NATURAL Essentials™, what you're reading may not be the current version. The latest version of the course is available for free download at www.spsimpson.com. It can be viewed and/or printed using Acrobat Reader, which can also be downloaded. Please compare the version number at the website with the one printed at the top of this page, ignoring the last (least significant) digit.

Related offerings

Consulting Services (Project Management, Systems Analysis & Design, Programming, Mentoring) . . . . . http://www.spsimpson.com

NATURAL Discussion Group . . . . . http://www.onelist.com/subscribe/NAT-U

(page intentionally blank)

Part 1
Essential Concepts

Part 1 - Essential Concepts

This part of the course introduces the essential concepts, primarily focusing on the different types of modules that make up a NATURAL application. Let's start by answering the obvious question ....

What is NATURAL?

NATURAL is a proprietary programming language developed and marketed by Software AG. Since it is a proprietary language, this course is primarily intended for technical people already part of the Software AG world and those who wish they were. Here's what they say about their product:

> *NATURAL, the Company's 4GL programming language for the enterprise environment, is designed to increase productivity in application software design, development and deployment. NATURAL supports Rapid Application Development to RDBMS environments with applications that are portable, scaleable and interoperable across multiple computing platforms*.
>
> *SAGA markets its products and services throughout North America, South America, Japan and Israel, and has global reach through access to Software AG's distribution channels in 50 countries.*
>
> *[source: SAGA 1997 Annual Report]*

Applications developed using NATURAL are modular, which contributes to its success as a tool for Rapid Application Development. In general, each module is created, maintained and stored independently of any others. While NATURAL lends itself to an object-oriented approach to systems development, this course does not attempt to teach object-oriented design - that's another subject in its own right.

This course was developed for version 2.2 of NATURAL.

The concepts introduced in this part of the course are as follows:

- programs
- subprograms
- subroutines
- in-line subroutines
- external subroutines
- copycode
- Program Editor
- data definitions
- local data
- Local Data Areas (LDAs)
- parameter data
- Parameter Data Areas (PDAs)
- global data
- Global Data Areas (GDAs)
- Data Area Editor
- maps
- windows
- Map Editor
- PF keys
- helproutines
- libraries
- Structured Mode vs. Reporting Mode
- database
- keys
- internal sequence number (ISN)
- repeating fields
- counters

Programs

The simplest type of module is a program. A program may be free-standing or may “call” lower level modules, such as subprograms and subroutines, that also contain executable code.

A NATURAL program can execute in either batch or on-line mode, with full access to database files. In on-line mode, NATURAL programs can process user input in real-time. In batch mode, NATURAL programs can read and write sequential (i.e. non-database) files.

Subprograms

Common logic (i.e. logic needed by multiple modules or multiple applications) is coded in subprograms so that it can be “called” as needed. For flexibility, parameter data can be passed directly between a subprogram and the “calling” routine. Using subprograms effectively avoids redundant code, so makes application maintenance easier.

For example you might write a subprogram to access a client file using a client id passed by the “calling” routine and return name and address information in a standard format. Different applications would “call” this subprogram to obtain correctly formatted data. If the formatting requirements changed, only the subprogram would need to be modified, not every module that handles client name and address information.

Subroutines

Subroutines are used to code discreet sub-functions to simplify logic and improve the readability and maintainability of the code.

There are 2 types of subroutines:

- in-line subroutines
- external subroutines

In-line subroutines

A program or subprogram will typically contain one or more in-line subroutines.

An in-line subroutine can only be “called” from within the module where it is coded. For this reason, the name need only be unique within that module. It’s possible to have several modules within the same application containing in-line subroutines with identical names. If such subroutines also have similar code, you should consider re-coding the logic as a subprogram, so it can be shared.

External subroutines

An external subroutine is the same as an in-line subroutine, except that it exists independently (in a module of type subroutine) and can be “called” from any other module. For this reason, the name must be unique within the application library.

One reason to create a subroutine module (i.e. a module containing a single external subroutine) is to remove code from a module that got too large to compile. By externalizing some of the in-line subroutines, the original module can be reduced in size.

The only other reason would be to create a common routine that can access pre-existing global data (i.e. data that is stored independently of any executable module, as explained in more detail later). Subprograms do not have access to such data. A common routine that doesn’t need access to pre-existing global data should be coded as a subprogram.

To give clients more flexibility, Software AG added the ability to pass parameter data between an external subroutine and the “calling” module. Parameter data cannot be passed to/from an in-line subroutine.

Copycode

Copycode modules contain lines of source code that can be invoked at “compilation time” into programs, subprograms or external subroutines by means of an in-line INCLUDE statement. Copycode may be parameterized with values from the INCLUDE statement being substituted at “compilation time”. The INCLUDE statement is covered in detail in part 2.

When designing applications, it’s common practice to provide a command level interface for the user, allowing them to access functions directly as an alternative to working through a series of menus. The logic to intercept a direct command and invoke the selected function is often written as copycode and included in every module that may need to process direct commands.

For anyone familiar with COBOL, copycode modules are equivalent to COBOL copybooks. One major difference is that in COBOL, copybooks are frequently used to define record layouts. In NATURAL, you would use local data areas (see below) instead.

Before creating a copycode module, consider whether a subprogram would be a better alternative.

Program Editor

Programs, subprograms, subroutines and copycode modules all represent executable code. NATURAL includes a special editor (i.e. the Program Editor) to create and maintain such modules. This is described in detail in appendix A.

Defining data

In NATURAL there are 2 ways to define data:

- in-line data definitions
- data definition modules

Later we’ll look at examples of in-line data definition. For now it’s enough to understand that in-line data definitions are part of the module (program, subprogram or external subroutine) where they are coded, and can not be referenced from other modules.

Data definition modules, on the other hand, are created and maintained independently and can be used by programs, subprograms and external subroutines as necessary. Not only is this more efficient, but use of a data area ensures that different modules contain identical data definitions, thus reducing the chance of error.

There are 3 types of data definition modules:

- local data areas (LDAs)
- parameter data areas (PDAs)
- global data areas (GDAs)

Local data

The term “local data” describes data that occupies space within the module where it is defined.

Local Data Areas (LDAs)

A local data area (LDA) defines data that becomes “local” to any module that uses the LDA. For example, if 2 modules reference the same LDA, there will be 2 parallel sets of data - one in each module. Data in one module will not be accessible from the other module.

Parameter data

The term “parameter data” describes data that is passed between a subprogram and the “calling” module. The data occupies space within the “calling” module only.

Parameter Data Areas (PDAs)

A parameter data area (PDA) is the preferred way to define parameter data within a subprogram.

Modules which “call” the subprogram can utilize its PDA as though it were a local data area (LDA) to define and reserve space for the data.

Consider the following example, ignoring any unfamiliar syntax:

Here’s a program which “calls” the subprogram:

```
** PROGRAM EXAMPLE
DEFINE DATA
LOCAL USING CLNT-PDA /* PDA FOR SUBPROGRAM CLIENT
.
END-DEFINE
.
.
CALLNAT ‘CLIENT’ CLNT-PDA
.
.
END /* EXAMPLE
```

Here's the subprogram which is "called":

```
** SUBPROGRAM CLIENT
DEFINE DATA
PARAMETER USING CLNT-PDA /* PDA FOR SUBPROGRAM CLIENT
.
END-DEFINE
.
.
END /* CLIENT
```

Notice that both modules reference the same parameter data area (PDA), thus eliminating the chance of inconsistent data definition.

Normally, every subprogram will have a unique parameter data area (PDA) associated with it. Note however that PDAs can be shared by multiple subprograms. Note also that a subprogram can reference multiple PDAs and that parameter data can be defined using in-line definitions within the subprogram.

Global data

The term "global data" describes data that occupies space outside of any executable module.

Programs and external subroutines (including any in-line subroutines within them) have direct access to global data, whereas subprograms and helproutines (including any in-line subroutines within them) do not. Any global data required by a subprogram or helproutine must be passed as a parameter.

Whenever a subprogram or helproutine module is invoked a new instance of global data can optionally be created, but access to the prior instance (if any) is not available. Programs and external subroutines invoked subsequently will have access only to the newly created instance (if any) and that will be discarded when the module that created it terminates. The prior instance, if any, will be unaffected.

Global Data Areas (GDAs)

A global data area (GDA) is the only way to define global data. Simply by referencing the GDA, an executable module (program, subprogram, external subroutine) has full access to the data.

An application, batch or on-line, can utilize a global data area (GDA) to store common data that is needed throughout the application. Since the data is stored in memory, access is very efficient.

Even though each executable module can reference only 1 GDA, an application may have multiple GDAs, but this is rare.

Data Area Editor

NATURAL includes a special editor (i.e. the Data Area Editor) to create and maintain LDA/PDA/GDA modules. This is described in detail in appendix A.

Maps

Map modules represent screen or report definitions.

Maps representing screen definitions are used in conjunction with an INPUT USING MAP statement (or REINPUT statement) to provide a way to view and manipulate data items inside a program, subprogram or subroutine. When interacting with a user, control stays in the map module until the user presses a PF key (see below) or ENTER. When control is returned to the ”calling” module, a system variable indicates which key was pressed.

When a map is used to define a screen layout, a one line message area is automatically reserved. This may be at the top or bottom of the screen, depending on installation parameters.

The syntax of the INPUT USING MAP and REINPUT statements provide a simple and effective way to populate the message area every time user input is required. Extensive and consistent use of this feature allows you to easily develop user-friendly (and rather “chatty”) applications.

A map defines a logical page which may be smaller than, the same size as, or bigger than the physical screen.

Windows

Windows allow you to view a logical page. A window is always present.

If the logical page is the same size as the physical screen, as is normally the case, the user will be unaware of the window.

If the logical page is bigger than the physical screen, multiple windows will be required to view it. If the logical page is smaller than the physical screen, it can be overlaid onto another window to create a “pop-up” effect; usually a frame is displayed around the window to exaggerate the effect. In these cases, the window must be explicitly defined using a DEFINE WINDOW statement, as described in part 2.

Logical pages bigger than the physical screen are generally avoided because they are confusing to the user.

Logical pages smaller than the physical screen are commonly used for “pop-up” windows, to provide a very intuitive way for users to “drill down” and get more detailed information. Effective use of this feature is an excellent way to make applications more user-friendly.

Map Editor

NATURAL includes a special editor (i.e. the Map Editor) to create and maintain maps. This is described in detail in appendix A.

PF keys

User input via a map is terminated by pressing ENTER or a function key (also known as a program function key, or PF key for short). These are so called because the function (i.e. purpose) of each key varies depending on the program. Generally they are known as PF1 through PF24 and PA1 through PA3.

In the NATURAL world, certain conventions apply. Some of the more obscure conventions are outside the scope of this course, but the essential ones are as follows:

- PF1 invokes help
- PF3 terminates
- PF7 scrolls up
- PF8 scrolls down

Following the conventions outlined above is highly recommended to provide a consistent user interface across different applications.

Since, by convention, PF3 is used to terminate an application, many developers use the nearby PF keys PF2 and/or PF4 for variations on that theme (e.g. terminate something less than the entire application). This is not recommended since users will often hit PF3 by accident, exit from the application and become frustrated.

Helproutines

A helproutine may take the form of a map module or a special kind of executable module, similar to a subprogram. Helproutines are used only in on-line applications and are only executed if invoked by the user. Invocation occurs when the user enters a help character (“?” by default) into the first character position of a map field or presses the help key (PF1 by convention).

A single application screen may have multiple helproutines associated with it - one to display screen-level help and one for each modifiable field to display field-level help. The screen-level helproutine will be invoked if the user presses the help key when the cursor is not positioned over a modifiable field. It will also be invoked if the user enters the help character into the first character position of a map field with no associated field-level help.

A single helproutine may be associated with multiple application screens. For example, a helproutine for client ID may be accessible from every screen which contains client ID as a modifiable field. The helproutine could allow the user to browse the entire client file to select the required client. This type of help is referred to as “active help” since the information returned to the user is not static (i.e. passive) in nature.

Giving users a dynamic selection capability greatly increases their control and makes for very user-friendly applications. With CONSTRUCT, Software AG’s NATURAL code generator, creating “active help” in the form of file browses is fast and simple.

"Active help" requires the helproutine to take the form of executable code. "Passive help" requires the helproutine to take the form of a map module containing static text such as "8-digit numeric code which uniquely identifies the client". Obviously, this is not as empowering as "active help".

When creating "passive help", give as much information as possible. If a user invokes help for "client ID" and learns nothing more than "code to identify client" they'll be frustrated, and rightly so. Knowing it's 8 characters numeric may help them understand the error message that caused them to invoke help in the first place.

Executable helproutine modules are maintained using the Program Editor. As mentioned, executable helproutine modules are similar to subprograms and carry parameters. An executable helproutine must have at least 1 parameter defined, either in-line or using a Parameter Data Area (PDA).

The mechanics of associating helproutines with application screens is covered in appendix A.

Libraries

NATURAL modules are arranged in libraries. Any number of libraries can exist and generally each application will have its own.

Common (e.g. cross-application) modules should be stored in a separate library which is "linked" to the other libraries. By default this is library SYSTEM.

Care should be taken when designing libraries. The number of modules a library can contain is limited, though not prohibitively so. Also, many of the tools provided with NATURAL work at the library, not cross-library, level.

Structured Mode vs. Reporting Mode

A long time ago, NATURAL evolved from what is now called Reporting Mode to Structured Mode. Although Reporting Mode is still available, it is outside the scope of this course.

If you ever find yourself in Reporting Mode by accident, type GLOBALS SM=ON to switch to Structured Mode.

Database

Throughout this course, I use the word database to refer to a system of managed data. NATURAL works with several different database products on multiple computing platforms.

Since my experience is with ADABAS (another Software AG product), I use the terms “file”, “record” and “field” rather than “table”, “row” and “column”, but beyond that, I’ve tried to keep the text generic.

Keys

Throughout this course, I use the term “key” to refer to a field (or collection of fields) used to access and sequence data. In NATURAL terminology, a single-element key is referred to as a “descriptor”. A multi-element key is referred to as a “superdescriptor”.

When a record is defined to ADABAS, or any other database system, it’s defined in terms of its component fields. Generally, each field has attributes associated with it, including an attribute to control how data is physically stored. In the ADABAS world, this is referred to as the “suppression” attribute. Most commonly “null suppression” is used, to minimize the amount of data space required.

For single-element keys where the descriptor field is null-suppressed, only data records with the descriptor field populated are indexed. For multi-element keys, every key-related null-suppressed field must be populated for the data record to be indexed. Stated another way, all null-suppressed elements of a key must be populated for the data record to be accessible via that key.

When designing keys, be sure to check that all elements of the key will be populated in every record.

Internal Sequence Number (ISN)

Within a database file, each record is assigned a unique identifier. In NATURAL, this is referred to as the Internal Sequence Number or ISN for short. When you access an existing record or add a new one, you can obtain the ISN and use it subsequently to re-access the record.

Accessing by ISN is more efficient than accessing by key and also safer, since key values do not have to be unique. When a record is accessed multiple times from within the same module, access the record by ISN as much as possible.

Since databases have to be reorganized from time to time, ISNs can change. For this reason, saving ISNs on database records or sequential files for any length of time should generally be avoided. If the database were to be reorganized, the stored ISNs could be incorrect, wreaking havoc.

Repeating fields

Some database products, including ADABAS, allow a field, or group of fields, to repeat multiple times. In NATURAL, a multiply occurring field on a database record is known as a “multiple field”, or MU for short. A multiply occurring group of fields is known as a “periodic group”, or PE for short.

Although a periodic group (PE) can include one or more multiple fields (MUs), this is generally considered bad file design.

The array handling features of NATURAL (described later) can be used to access and manipulate the data in both multiple fields (MUs) and periodic groups (PEs).

Counters

For repeating fields, or groups of fields, a counter can be interrogated to determine the number of occurrences that are populated. If any field within a group is populated, the group is considered to be populated. The name of the counter is *C*name* where name represents the name of the repeating field or group. Counters are dynamically maintained by the system and cannot be changed programmatically. Later in the course you will see an example.

Summary

NATURAL concepts		
Concept	**Symbol**	**Description**
Program	P	Executable code that alone or with other modules constitutes an application.
Subprogram	N	Callable module for common logic that does not need access to global data.
In-line Subroutine		Discreet logic that can only be executed from within the module where it is coded.
External Subroutine	S	Discreet logic that exists independently of (and can be “called” from) any executable module. Used for common routines that need access to global data.
Copycode	C	Source code which can be included at compile time.
Program Editor		Tool for creating and maintaining the source of executable modules.
Local Data		Data that occupies space within the module where it is defined.
Local Data Area	L	Common data definition. The defined data occupies space in every module that uses the LDA.
Parameter Data		Data that is passed to, or returned by, a subprogram or helproutine. The defined data occupies space in the “calling” module only.
Parameter Data Area	P	Parameter data definition for a subprogram or helproutine. “Calling” modules may use the PDA as an LDA.
Global Data		Data that occupies space which is independent of any executable module.
Global Data Area	G	Global data definition.
Data Area Editor		Tool for creating and maintaining the source of data definition modules.
Map	M	Screen definition, report definition or help text.
Map Editor		Tool for creating and maintaining map modules.
PF Keys		Keys which are used to terminate user input.
Helproutine	H	Module containing executable logic to provide screen-level or field-level help on demand.
Library		Collection of related modules, usually representing a single application.
Database		A system of managed data.
Key		A field or collection of fields used to access and sequence data.
ISN		Internal Sequence Number. A number that uniquely identifies a record within a database file.
Repeating field		A field, or group of fields, that repeats multiple times.
Counter		A system-maintained field indicating how many occurrences of a repeating field, or group of fields, are populated.

Note the symbols (e.g. P for program, L for LDA, etc.) in the second column. These are used when creating new modules, to indicate the required module type.

(page intentionally blank)

Part 2
Essential Syntax

Part 2 – Essential Syntax

This part of the course covers much of the syntax of NATURAL and is arranged as follows:

- Structure of executable modules
- Defining data
- Data manipulation
- Input and output
- Flow control
- Database access
- Transaction control
- Database query
- Sequential files
- Exiting a routine or loop
- System variables and functions
- Syntax summaries

The syntax summaries include some NATURAL statements and variations not covered in this part of the course.

Structure of executable modules

Executable modules contain data definitions, control logic, some in-line subroutines (usually) and are terminated with an END statement. They may also contain comments and/or INCLUDE statements to invoke copycode.

Program, subprogram and helproutine modules

The following diagram illustrates the structure of program, subprogram and helproutine modules.

Data definition

```
DEFINE DATA
.
END-DEFINE
```

Control logic and in-line subroutines

```
READ …
  .
  IF …
    PERFORM …
    .
  END-IF
  .
  .
END-READ
*
DEFINE SUBROUTINE …
.
END-SUBROUTINE
*
DEFINE SUBROUTINE …
.
END-SUBROUTINE
*
```

END statement

```
END
```

By convention, any in-line subroutines are placed immediately before the END statement, as illustrated above. They could be placed anywhere between END-DEFINE and END, i.e. before, after, or even interspersed with, the control logic.

The name “in-line” subroutine refers only to placement. Subroutines are never executed in-line - they must be explicitly “called”. In-line subroutines cannot be nested one inside another.

Subroutine modules

Subroutine modules (i.e. modules containing a single external subroutine) have a slightly different structure, as illustrated below.

Data definition

```
DEFINE DATA
.
END-DEFINE
```

Control logic and in-line subroutines

```
DEFINE SUBROUTINE CALC-INTEREST
READ …
  .
  IF …
    PERFORM …
    .
  END-IF
  .
  .
END-READ
*
DEFINE SUBROUTINE …
.
END-SUBROUTINE
*
DEFINE SUBROUTINE …
.
END-SUBROUTINE
*
END-SUBROUTINE /* CALC-INTEREST
```

END statement

```
END
```

The only difference is that the control logic must be coded as a subroutine, i.e. start with DEFINE SUBROUTINE and end with END-SUBROUTINE. Any in-line subroutines must be placed between the two. Again, the convention is to code in-line subroutines at the end.

The name “external” subroutine refers to placement – in a module other than the one from which it is “called”.

Data definitions

Data definitions start with DEFINE DATA and terminate with END-DEFINE, as described below. All statements related to local data, parameter data and global data are contained between the two. These are described in detail in the section which follows.

Control logic and in-line subroutines

The control logic of a module immediately follows the data definitions. This logic should be as simple as possible, with subfunctions broken out and coded as in-line subroutines. Subroutines, whether in-line or external, start with DEFINE SUBROUTINE and terminate with END-SUBROUTINE, as described below. All but the simplest of executable modules will contain at least one in-line subroutine.

END statement

An END statement is used to mark the physical end of a module.

Comments

In executable modules and copycode, source lines with “**” (double asterisk) or “* “ (asterisk space) in the first 2 character positions are treated as comments, i.e. ignored. Also, characters occurring after “/*” (slash asterisk) on any source line are treated as comments.

In data definition modules (LDA, PDA and GDA), source lines with “*” (asterisk) in the first (Type) column are treated as comments. Also, if the entry in the last (Index/Init/EM/Name/Comment column) starts with “/*” (slash asterisk) it is treated as a comment.

INCLUDE statement

An INCLUDE statement invokes source lines from a copycode module at compile time. Consider the following example, ignoring any unfamiliar syntax:

Here's a module which contains an INCLUDE statement:

```
.
DEFINE SUBROUTINE POPULATE-AUDIT-FIELDS
************************************************************
** Updates audit fields on file (UPDATE-VIEW) and map.
************************************************************
INCLUDE IVAUDTC1 'UPDATE-VIEW'
*
MOVE EDITED UPDATE-VIEW.AUDIT-TIME(EM=MM/DD/YYYY' 'HH:II:SS)
  TO #MAP.#AUDIT-TIME
#MAP.AUDIT-PROGRAM := UPDATE-VIEW.AUDIT-PROGRAM
#MAP.AUDIT-USER := UPDATE-VIEW.AUDIT-USER
END-SUBROUTINE /* POPULATE-AUDIT-FIELDS
.
```

Here's the copycode which is invoked:

```
** COPYCODE IVAUDTC1 ************************************ BEGIN
&1&.AUDIT-TIME := *TIMX
&1&.AUDIT-PROGRAM := *PROGRAM
&1&.AUDIT-USER := *USER
ADD 1 TO &1&.AUDIT-COUNTER
** COPYCODE IVAUDTC1 ************************************** END
```

Here's the source code which is presented at compile time:

```
.
DEFINE SUBROUTINE POPULATE-AUDIT-FIELDS
************************************************************
** Updates audit fields on file (UPDATE-VIEW) and map.
************************************************************
** COPYCODE IVAUDTC1 ************************************* BEGIN
UPDATE-VIEW.AUDIT-TIME := *TIMX
UPDATE-VIEW.AUDIT-PROGRAM := *PROGRAM
UPDATE-VIEW.AUDIT-USER := *USER
ADD 1 TO UPDATE-VIEW.AUDIT-COUNTER
** COPYCODE IVAUDTC1 ************************************* END
*
MOVE EDITED UPDATE-VIEW.AUDIT-TIME(EM=MM/DD/YYYY' 'HH:II:SS)
  TO #MAP.#AUDIT-TIME
#MAP.AUDIT-PROGRAM := UPDATE-VIEW.AUDIT-PROGRAM
#MAP.AUDIT-USER := UPDATE-VIEW.AUDIT-USER
END-SUBROUTINE /* POPULATE-AUDIT-FIELDS
.
```

Notice that the value "UPDATE-VIEW" is specified as a parameter in the INCLUDE statement to replace the "&1&" value in the copycode.

Up to 98 values can be specified as parameters in an INCLUDE statement. Each value is substituted for a placeholder in the form "&1&" through "&98&".

DEFINE SUBROUTINE statement

A DEFINE SUBROUTINE statement introduces a subroutine. The subroutine ends with an END-SUBROUTINE statement. Consider the following example, ignoring any unfamiliar syntax:

```
.
DEFINE SUBROUTINE POPULATE-AUDIT-FIELDS
************************************************************************
** Updates audit fields on file (UPDATE-VIEW) and map.
************************************************************************
.
END-SUBROUTINE /* POPULATE-AUDIT-FIELDS
.
```

It's good practice to document the purpose of each subroutine by placing a brief comment at the start, as in the above example. I like to surround the comments with a "flower box", like the one above, to increase readability.

The sequence is not significant, but I like to arrange my subroutines in alphabetical order, to make them easy to find.

Data definition

DEFINE DATA statement

Data definition statements start with DEFINE DATA and terminate with END-DEFINE. Consider the following examples:

Example 1: In-line data definition

```
DEFINE DATA
.
LOCAL
1 #SUBTOTAL (P7.2)
.
END-DEFINE
.
```

LOCAL followed by in-line data definition statements is the usual way to define variables which don't need to be referenced outside of the module, for example fields holding intermediate results of some kind.

Example 2: Data definition using an LDA

```
DEFINE DATA
.
LOCAL USING CLNT-PDA /* PDA FOR SUBPROGRAM CLIENT
LOCAL
1 #SUBTOTAL (P7.2)
.
END-DEFINE
.
.
```

LOCAL USING allows you to reference an LDA or PDA defining local data.

This example illustrates how in-line and LDA definitions can be mixed-and-matched using a combination of LOCAL and LOCAL USING statements.

Example 3: Data definition using a PDA

```
DEFINE DATA
PARAMETER USING CLNT-PDA /* PDA FOR SUBPROGRAM CLIENT
.
LOCAL
1 #INDEX (P3)
.
END-DEFINE
.
```

PARAMETER USING allows you to reference a PDA (defining parameter data) from within a subprogram (or helproutine).

You can mix-and-match PARAMETER and PARAMETER USING statements, but this is not recommended.

In this example, the subprogram also has some local data defined in-line.

Example 4: Data definition using a GDA

```
DEFINE DATA
GLOBAL USING INVDATA /* GDA FOR INVOICING APPLICATION
LOCAL USING INVREDEF /* LDA REDEFINING INVOICING GDA
.
END-DEFINE
.
```

GLOBAL USING allows you to reference a GDA defining global data. For global data there is nothing equivalent to in-line definition.

In this example, the program also has some local data defined using an LDA.

Data formats

At the lowest level data definition statements, whether in-line or contained in an LDA, PDA or GDA, define elementary fields. Each has a pre-defined format and length. The essential data formats are as follows:

Data formats

Format	Max Length	Bytes	Description
A	253	length	Alpha (letters, numbers, certain symbols)
N	29	length	Numeric
I	4	1, 2 or 4	Integer
P	29	(length /2) + 1	Packed
B	126	length	Binary
D		4	Date (internal day number)
T		7	Time (internal day number and time)
L		1	Logical (TRUE or FALSE)
C		1	Control (whether modified)

Record structures

At a higher level, data definition statements, whether in-line or in LDA/PDA/GDA form, are used to define record structures. A record structure is simply a group of related elementary fields. Consider the following examples:

Example 1: 2-level record structure

```
DEFINE DATA
.
LOCAL
1 CLNT-REC /* CLIENT RECORD
  2 CLIENT-ID (N8)
  2 LAST-NAME (A20)
  2 FIRST-NAME (A20)
  2 MIDDLE-INITIAL (A1)
.
END-DEFINE
.
```

Notice that format and length is not specified for CLNT-REC because it is not an elementary field. CLIENT-ID, LAST-NAME, etc. are recognized as elementary fields because they have a format and length specified.

When working with record structures, it's good practice to "qualify" field names by prefixing them with the group name followed by a period; for example "CLNT-REC.CLIENT-ID" and "CLNT-REC.LAST-NAME", etc.

When a module has several record structures with common elementary field names, qualifying the field names is mandatory.

Indentation to reflect the structure is optional; 2 character positions per level is recommended.

Example 2: 3-level record structure

```
*                                        PARAMETER DATA AREA CLNT-PDA
  1 CLNT-PDA
  2 INPUT-PARAMETERS
  3 CLIENT-ID                    N   8
  2 OUTPUT-PARAMETERS
  3 LAST-NAME                    A  20
  3 FIRST-NAME                   A  20
  3 MIDDLE-INITIAL               A   1
.
```

In this example, an extra level is used to show which fields represent input data and which represent output data. Another possibility is INPUT-OUTPUT-PARAMETERS, i.e. data which is required on input and may be changed on output. I recommend this technique as a way to document your logic; anyone coding a module to “call” the subprogram immediately knows what to expect.

Notice that format and length is not specified for INPUT-PARAMETERS because it is not an elementary field. The same applies to OUTPUT-PARAMETERS.

Qualified field names should contain only the highest and lowest level elements, in the above example CLNT-PDA.CLIENT-ID would be valid, CLNT-PDA.INPUT-PARAMETERS.CLIENT-ID would not.

Indentation is not possible with LDAs, PDAs and GDAs.

Database views

Database fields are inherently organized into records. In NATURAL, access to database fields is by means of a view. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 UPDATE-VIEW VIEW OF CLIENT
  2 CLIENT-ID
  2 BILLING-NAME
  2 BILLING-ADDRESS (1:3)
  2 AUDIT-TIME
  2 AUDIT-USER
  2 AUDIT-PROGRAM
  2 AUDIT-COUNTER
.
END-DEFINE
.
.
```

In this example a view of CLIENT is used.

The view name must be a valid name within the database system. Format and length should not be specified - they will be derived directly from the database system. Only those fields referenced in the module need to be defined.

The Program Editor has a special .G (generate) command which can be used to generate database views. This is described in appendix A.

Views may also be used in an LDA/PDA/GDA. Consider the following example:

```
V 1 UPDATE-VIEW                      CLIENT
  2 CLIENT-ID                  N   8
  2 BILLING-NAME               A  40
  2 BILLING-ADDRESS            A  40 (1:3)
  2 AUDIT-TIME                 T
  2 AUDIT-USER                 A   8
  2 AUDIT-PROGRAM              A   8
  2 AUDIT-COUNTER              P   7
.
```

The Data Area Editor has a special .V (view) command for selecting fields from a database view. Fornat and length are derived automatically. This is described in appendix A.

Redefining data

An elementary field can be redefined using a REDEFINE statement. Consider the following examples.

Example 1: In-line redefinition

```
DEFINE DATA
.
LOCAL
1 CLNT-REC /* CLIENT RECORD
  2 CLIENT-ID (N8)
  2 LAST-NAME (A20)
  2 REDEFINE LAST-NAME
    3 FIRST-LETTER-OF-LAST-NAME (A1)
  2 FIRST-NAME (A20)
  2 MIDDLE-INITIAL (A1)
.
END-DEFINE
.
```

This example illustrates how REDEFINE is used with in-line data definitions.

The REDEFINE statement must have the same level number as the base field. As illustrated above, the newly defined field may be shorter than the base field.

Example 2: LDA/PDA/GDA redefinition

```
*                                          PARAMETER DATA AREA CLNT-PDA
  1 CLNT-PDA
  2 INPUT-PARAMETERS
  3 CLIENT-ID                    N   8
R 3 CLIENT-ID
  4 CLIENT-ID-ALPHA              A   8
  2 OUTPUT-PARAMETERS
  3 LAST-NAME                    A  20
R 3 LAST-NAME
  4 FIRST-LETTER-OF-LAST-NAME    A   1
  3 FIRST-NAME                   A  20
  3 MIDDLE-INITIAL               A   1
.
.
```

This example illustrates how REDEFINE is used with a PDA. The example could just as easily be an LDA or GDA.

In this example, CLIENT-ID is also redefined. Redefining a numeric field as alpha in this way, allows you to reference the data with leading zeros; this topic will be expanded later in the course.

Initializing data

By default, fields start out with a null value. For alphanumeric (format A) fields this is a space. For numeric fields (format N, P, B or I) this is zero. For date and time field (formats D and T) this is a null date and time. For logical fields (format L) this is a value of FALSE.

INIT clause

An initial value (one which can be subsequently changed by the executing logic) can be assigned using an INIT clause. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #DAYS (P3) INIT<60>
1 #MESSAGE (A50) INIT<'WE APPRECIATE YOUR BUSINESS'>
1 #FOLLOW-UP (L) INIT<TRUE>
.
END-DEFINE
.
```

For alphanumeric fields, the initial value must be enclosed in single quotes.

You cannot assign an initial value to a field defined using a REDEFINE statement, but you can assign an initial value to the "base" field. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #MESSAGE (A50) INIT<'WE HOPE YOU ENJOY'>
1 REDEFINE #MESSAGE
  2 #ENJOY (A18)
  2 #PRODUCT-NAME (A32)
.
END-DEFINE
.
```

CONST clause

A constant value (one which cannot be changed by the executing logic) can be assigned using a CONST clause. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #DAYS (P3) CONST<60>
1 #MESSAGE (A50) CONST<'WE APPRECIATE YOUR BUSINESS'>
1 #SUPPRESS-FOLLOW-UP (L) CONST<TRUE>
.
END-DEFINE
.
```

Arrays

An array is simply multiple occurrences of a single field, or group of fields. 1, 2 and 3 level arrays are allowed. Consider the following examples, ignoring any unfamiliar syntax:

Example 1: In-line 1-level array

```
DEFINE DATA
LOCAL
.
1 #SHORT-NAME (A3/1:12) CONST<'JAN','FEB','MAR','APR','MAY',
'JUN','JUL','AUG','SEP','OCT','NOV','DEC')
.
1 #MONTH-NBR (P2)
.
END-DEFINE
.
.
#MONTH := #SHORT-NAME(#MONTH-NBR)
.
```

In this example, the notation "/1:12" after the format and length definition defines 12 occurrences of the field. Each can be referenced by an index in the range 1 to 12. Initial or constant values can be assigned if appropriate, using the syntax shown above.

Below the END-DEFINE statement is an example of logic to extract a value from the array using field #MONTH-NBR as the index. When the logic is executed, an execution time error will occur if #MONTH-NBR contains a value outside the range of the array (1 through 12).

Example 2: In-line 2-level array

```
DEFINE DATA
LOCAL
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5,1:12) /* 5 DEPARTMENTS, 12 MONTHS
1 #MONTH-NBR (P2)
1 #TOTAL (P7.2)
.
END-DEFINE
.
.
ADD #DOLLARS(#DEPT-NBR,#MONTH-NBR) TO #TOTAL
.
```

In this example, the notation “/1:5,1:12” after the format and length definition defines 5 sets of 12 occurrences of the field. Each one is referenced by 2 indexes. In this example, the first index (representing the department) must be in the range 1 to 5; the second index (representing the month) must be in the range 1 to 12.

Below the END-DEFINE statement is an example of logic to add a value from the array (using fields #DEPT-NBR and #MONTH-NBR as indexes) into another variable.

Example 3: 2-level array within an LDA

```
1 #DOLLARS                    P 7.2  (1:5,1:12)
```

In this example, the notation “(1:5,1:12)” after the format and length definition defines 5 sets of 12 occurrences of the field. Although this notation is slightly different from example 2, the meaning is the same.

Example 4: Structured in-line 1 level array

```
DEFINE DATA
.
LOCAL
1 #CLIENT-ID (N8)
.
1 CLNT-REC (1:2) /* CLIENT RECORD, 2 OCCURRENCES
  2 CLIENT-ID (N8)
  2 LAST-NAME (A20)
  2 REDEFINE LAST-NAME
    3 FIRST-LETTER-OF-LAST-NAME (A1)
  2 FIRST-NAME (A20)
  2 MIDDLE-INITIAL (A1)
.
END-DEFINE
.
#CLIENT-ID := CLNT-REC.CLIENT-ID(1)
.
```

In this example, the notation “(1:2)” after the group name defines 2 occurrences of the group. Each elementary field within the group must be referenced by an index in the range 1 to 2.

Below the END-DEFINE statement is an example of logic to extract a value from the array using a hard-coded index value of 1.

An array can contain REDEFINE statements as illustrated in example 4 above. Arrays may also be redefined, as illustrated in the following example:

```
DEFINE DATA
LOCAL
.
1 #SHORT-NAME (A3/1:12) CONST<‘JAN’,,MAR,,,JUN,,,,,,’DEC’>
1 REDEFINE #SHORT-NAME
  2 #STRING (A36)
.
END-DEFINE
.
```

In this example, #STRING would contain “JAN MAR JUN DEC“.

Edit masks

Edit masks are used to validate and format data. For validation, they are used in conjunction with IF statements and certain other statements. For formatting they are used in conjunction with MOVE EDITED, DISPLAY, WRITE and certain other statements. They are also used in map definitions to format output.

Although these statements are described in some detail later, edit masks are not. The essential edit masks are described here for future reference.

Edit masks for validating input		
Symbol	**Description**	**Example**

9	numeric character
	in specified position
		#COUNT =
		MASK(999999)

A	alpha character in specified position	#STATE = MASK(AA)
L	lower case alpha character in specified position	#NAME = MASK (LLLLLLLLLLLLLL)
U	upper case alpha character in specified position	#NAME = MASK (UUUUUUUUUUUUUU)
C	alpha character, numeric character or space in specified position	#APT = MASK(CCCCC)
P	printable character in specified position	#NAME = MASK (PPPPPPPPPPPPPP)
. or ? or _ (underscore)	any character in specified position	#NAME = MASK(....'X')
'X'	specified character in specified position	#NAME = MASK(....'X')

H	hexadecimal character (A-F or 0-9)
	in specified position
		#HEX-VALUE =
		MASK(HH)

MM	month of year	#DATE =
		MASK(MMDDYYYY)

DD	day of month	
YY	year (last 2 digits)	
YYYY	year (all 4 digits)	

HH	hour of day	#TIME =
		MASK(HHIISS)

II	minute of hour	
SS	second of minute	

Avoid writing code to test date elements separately, since the range of valid values for some elements varies depending on the context. When DD is tested separately, the current month (at execution time) provides the context, so a value of 31 is considered valid when the logic is executed during March, but not when the exact same logic is executed during April. When the logic is executed during February, a value of 29 will only be considered valid if the current year (at execution time) is a leap year.

Edit masks for formatting output		
Symbol	**Description**	**Example**
999999	display digits	#COUNT(EM=999999)
ZZZZZ9	suppress digits	#COUNT(EM=ZZZZZ9)
9-	floating sign (trailing, - if negative)	#VALUE(EM=9-)
-9	floating sign (leading, - if negative)	#VALUE(EM=-9)
9+	floating sign (trailing, + or -)	#VALUE(EM=9+)
+9	floating sign (leading, + or -)	#VALUE(EM=+9)
.	decimal point alignment	#PRICE(EM=ZZ9.99)
'/'	(any literal character or string)	#DATE (EM=MM'/'DD '/'YYYY)
MM	month of year	
DD	day of month	
YY	year (last 2 digits)	
YYYY	year (all 4 digits)	
JJJ	julian (day of year)	#DATE (EM=YYJJJ)
LLLLLLLLL	name of month (e.g. SEP)	#DATE (EM=LLL)
O	day of week - number (day 0 is defined by local parameter, language dependent)	#DATE (EM=O)
NNNNNNNNN	day of week - name (e.g. WED, day 0 is defined by local parameter, language dependent)	#DATE (EM=NNN)
HH	hour (of day)	#TIME (EM=HH':'II':' SS'.'T 'AP)
II	minute (of hour)	
SS	second (of minute)	
T	tenths of seconds	
AP	AM or PM (without, yields military time)	
F/T	T (true) or F (false)	#FIRST (EM=F/T)
N/Y	N (no) or Y (yes)	#FIRST (EM=N/Y)
H	hexadecimal	#ALPHA3(EM=HHH)

Where possible, avoid 2-digit years. If the target field implicitly or explicitly requires a 4-digit year, the current century (at execution time) may be assumed. D format variables, for example, implicitly require a 4-digit year. To provide flexibility for the year 2000, a software patch (a.k.a. zap) is available from Software AG which causes the century to be determined based on the 2-digit year and a site-specific table defining the range for each century.

Data manipulation

Statements in this category manipulate data. The following are covered in this section of the course:

- RESET
- := (becomes equal to)
- ADD
- SUBTRACT
- MOVE
- MOVE ALL
- COMPRESS
- EXAMINE
- EXAMINE TRANSLATE

RESET statement

RESET assigns a null value to elementary fields.

For alphanumeric (format A) fields this is a space. For numeric fields (format N, P, I or B) this is zero. For date and time field (formats D and T) this is a null date and time. For logical fields (format L) this is a value of FALSE.

By default, fields start out with a null value, so an initial RESET is unnecessary.

Consider the following examples:

Example 1: Resetting individual fields

```
DEFINE DATA
LOCAL
.
1 #LAST-NAME (A20)
1 #FIRST-NAME (A20)
.
END-DEFINE
.
.
RESET
  #LAST-NAME
  #FIRST-NAME
.
```

Note that multiple fields can be reset using a single RESET statement.

Example 2: Resetting array entries

```
DEFINE DATA
LOCAL
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5,1:12) /* 5 DEPARTMENTS, 12 MONTHS
1 #MONTH-NBR (P2)
.
END-DEFINE
.
.
RESET #DOLLARS(#DEPT-NBR,1:6)
.
.
RESET #DOLLARS(4,*)
.
.
RESET #DOLLARS(*,3)
.
.
RESET #DOLLARS(#DEPT-NBR,#MONTH-NBR)
.
```

When resetting an array, specific elements can be specified as illustrated above. “All” elements is represented by an asterisk, also illustrated above.

Example 3: Resetting at the group level

```
DEFINE DATA
.
LOCAL
1 CLNT-REC /* CLIENT RECORD
  2 CLIENT-ID (N8)
  2 LAST-NAME (A20)
  2 FIRST-NAME (A20)
  2 MIDDLE-INITIAL (A1)
.
END-DEFINE
.
.
RESET CLNT-REC
.
```

In this example, RESET is specified for a “group field”. All associated elementary fields are individually reset.

:= (becomes equal to) statement

There are several direct assignment statements, including ASSIGN, COMPUTE and MOVE, but := (becomes equal to) is the quickest. The “target” field is specified before the ":=“ symbol. The “source” field is specified after. The “source” field can be a literal, a variable or a formula.

Consider the following examples:

Example 1: Assignment from another variable

```
DEFINE DATA
LOCAL
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5,1:12) /* 5 DEPARTMENTS, 12 MONTHS
1 #INVOICE-AMT(P7.2)
1 #MONTH-NBR (P2)
.
END-DEFINE
.
.
#DOLLARS(#DEPT-NBR,#MONTH-NBR) := #INVOICE-AMT
.
```

Example 2: Literal assignment

```
DEFINE DATA
LOCAL
.
1 #MAX-CLIENT (N8)
1 #MIN-CLIENT (N8)
.
END-DEFINE
.
.
#MIN-CLIENT := 1
#MAX-CLIENT := 99999999
.
```

In this example, literal values are assigned to variables.

Example 3: Hexadecimal literal assignment

```
DEFINE DATA
LOCAL
.
1 #MAX-KEY (A8)
.
END-DEFINE
.
.
#MAX-KEY := H'FFFFFFFFFFFFFFFF'
.
```

In this example, a literal value is assigned to a variable.

The letter H in front of the literal indicates a hexadecimal value is being assigned. There must be an even number of valid hexadecimal characters in the literal string.

When assigning a hexadecimal value, be sure the literal string contains the correct number of digits. A short literal will cause the target variable to be padded with spaces, causing unexpected results. See MOVE ALL below.

Example 4: Arithmetic assignment

```
DEFINE DATA
LOCAL
.
1 #INV-DUE-DATE (D)
.
END-DEFINE
.
.
#INV-DUE-DATE := #INV-DUE-DATE + 30
.
```

In this example, the “source field” is a formula which increases a date by 30 days. Note that NATURAL date (format D) fields can be manipulated in this way.

The formula may be simple as in the above example or more elaborate. The following table shows the available arithmetic operators:

Arithmetic operators	
Symbol	**Meaning**
+ (plus)	add
- (hyphen)	subtract
* (asterisk)	multiply
/ (slash)	divide

Formulas are evaluated using standard mathematical rules, for example:

```
#ELEVEN := #THREE * #FOUR - #ONE
```

Parentheses can be used to control the evaluation, for example:

```
#NINE := #THREE * (#FOUR - #ONE)
```

COMPUTE statement

The COMPUTE statement is another direct assignment statement. COMPUTE must be used when rounding is required, for example:

```
COMPUTE ROUNDED #SIX = #ELEVEN / #TWO
```

Assuming the variable #SIX is defined without decimal places, the result (5.5) would be rounded up to 6, per standard mathematical rules for rounding.

Contrast the above example with a COMPUTE statement without the ROUNDED clause:

```
COMPUTE #FIVE = #ELEVEN / #TWO
```

Assuming the variable #FIVE is defined without decimal places, the result (5.5) would be truncated to 5.

ADD statement

The ADD statement can be used to change the value of a variable. Consider the following example:

```
ADD 30 TO #DUE-DATE
```

Although redundant to := (becomes equal to), an ADD statement increases the readability of your code. Compare the example above with the alternative:

```
#DUE-DATE := #DUE-DATE + 30
```

SUBTRACT statement

The SUBTRACT statement can be used to change the value of a variable. Consider the following example:

```
SUBTRACT 30 FROM #DUE-DATE
```

MOVE statement

The MOVE statement has several forms. The essential ones are as follows:

- MOVE EDITED
- MOVE BY NAME
- MOVE LEFT
- MOVE RIGHT

There is also a MOVE ALL statement.

MOVE statement - MOVE EDITED

MOVE EDITED allows you to format data using edit masks. Consider the following examples:

Example 1: Formatting a number

```
DEFINE DATA
LOCAL
.
1 #INVOICE-AMT (P7.2)
.
1 #PRINT-LINE
  2 #INV-AMT (N7.2)
.
END-DEFINE
.
.
MOVE EDITED #INVOICE-AMT (EM=ZZZZZZ9.99-)
  TO #PRINT-LINE.#INV-AMT
.
```

In this example, the MOVE EDITED statement formats a number for output. An edit mask is used to suppress leading zeros, insert a decimal point and a trailing minus sign if the value is negative.

Example 2: Formatting a date

```
DEFINE DATA
LOCAL
.
1 #DUE-DATE (D)
.
1 INVOICE-REC
  2 CLIENT-ID (N8)
.
  2 DUE-DATE (A8) /* MMDDYYYY
.
END-DEFINE
.
.
MOVE EDITED #DUE-DATE (EM=MMDDYYYY) TO INVOICE-REC.DUE-DATE
.
```

In this example, the MOVE EDITED statement formats a date for output. An edit mask is used to produce a date in MMDDYYYY format.

Time variables (format T) contain both a date and a time. You can extract a date, a time or both. Example 2 above would work equally well with a time variable.

The ability to do arithmetic on dates and easily convert the result using MOVE EDITED, as illustrated in the prior examples, is very useful.

You may find it helpful to re-read the section on edit masks for formatting data, and review those edit masks which can be used in conjunction with the MOVE EDITED statement.

MOVE statement - MOVE BY NAME

MOVE BY NAME copies data from one record structure to another. Every elementary field with an identical name in both structures is copied.

If the format and/or length is different, the data is converted automatically. Any arrays common to both record structures must have an identical number of occurrences.

Consider the following example:

```
DEFINE DATA
LOCAL
.
1 INVOICE-REC
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 PRODUCT-ID (N6)
  2 PRODUCT-NAME (A20)
  2 QUANTITY (N3)
  2 TOTAL-PRICE (N7.2)
  2 NBR-INSTALLMENTS (N2)
  2 DUE-AMOUNT (N7.2/1:12)
  2 TOTAL-DUE (N7.2)
  2 DUE-DATE (A8) /* MMDDYYYY
  2 SALES-MESSAGE (A40/1:5)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
  2 PRODUCT-ID
  2 QUANTITY
  2 TOTAL-PRICE
  2 PAID-AMOUNT
  2 C*BILL-DATE (1:12)
  2 BILL-DATE (1:12)
  2 UNPAID-FLAG
  2 AUDIT-TIME
  2 AUDIT-PROGRAM
  2 AUDIT-USER
  2 AUDIT-COUNTER
.
END-DEFINE
.
.
MOVE BY NAME RECEIVABLE TO INVOICE-REC
.
```

MOVE statement - MOVE LEFT

MOVE LEFT copies data into an alpha field, deleting any leading spaces so that the data is left-justified. Consider the following example:

```
MOVE LEFT #LAST-NAME TO #LAST-NAME
```

MOVE statement - MOVE RIGHT

MOVE RIGHT copies data into an alpha field, inserting leading spaces as necessary so that the data is right-justified. Consider the following example:

```
MOVE RIGHT LAST-NAME TO #LAST-NAME
```

MOVE ALL statement

MOVE ALL fills the target variable with a literal string, repeated as necessary. Consider the following examples:

```
MOVE ALL '?' TO #LAST-NAME
```

```
MOVE ALL H'FF' TO #MAX-KEY
```

COMPRESS statement

COMPRESS joins 2 or more strings. Consider the following examples:

Example 1: COMPRESS

```
DEFINE DATA
LOCAL
.
1 #ENJOY (A25) CONST<‘WE HOPE YOU ENJOY’>
1 #MESSAGE (A33)
1 #PRODUCT-NAME (A25)
.
END-DEFINE
.
.
#PRODUCT-NAME := ‘PAYING TOO MUCH ATTENTION’
.
COMPRESS #ENJOY #PRODUCT-NAME INTO #MESSAGE
.
```

The COMPRESS statement removes any superfluous spaces between each string, leaving only one. In this example, the resulting string would be "WE HOPE YOU ENJOY PAYING TOO MUCH” - perhaps not the right message for an invoice.

Example 2: COMPRESS LEAVING NO SPACE

```
DEFINE DATA
LOCAL
.
1 #CLIENT-ID (N8)
1 REDEFINE #CLIENT-ID
  2 #CLIENT-ID-ALPHA (A8)
.
1 #DEPT-NBR (P1)
1 #MESSAGE (A33)
.
END-DEFINE
.
.
#CLIENT-ID := 31722
.
#DEPT-NBR := 4
.
COMPRESS #DEPT-NBR '-' #CLIENT-ID-ALPHA
  INTO #MESSAGE LEAVING NO SPACE
.
```

In this example, the strings will be joined leaving no spaces and leading zeros in client ID will be retained. The resulting string would be "4-00031722".

EXAMINE statement

The EXAMINE statement has several forms. The essential ones are as follows:

- EXAMINE REPLACE
- EXAMINE DELETE
- EXAMINE GIVING LENGTH
- EXAMINE GIVING POSITION
- EXAMINE GIVING INDEX

EXAMINE statement - EXAMINE REPLACE

EXAMINE REPLACE finds and replaces part of a string. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #PRODUCT-NAME (A25)
.
END-DEFINE
.
.
#PRODUCT-NAME := 'PAYING TOO MUCH ATTENTION'
.
EXAMINE #PRODUCT-NAME FOR 'MUCH' REPLACE WITH 'LITTLE'
.
```

In this example the resulting value in #STRING would be "PAYING TOO LITTLE ATTENTION".

If the search string occurs multiple times, by default all occurrences will be replaced. To replace only the first, specify REPLACE FIRST. Consider the following example:

```
EXAMINE #PRODUCT-NAME FOR 'MUCH' REPLACE FIRST WITH 'LITTLE'
```

In all forms of EXAMINE, trailing spaces are considered insignificant unless EXAMINE FULL is specified. In EXAMINE REPLACE, trailing spaces in the replacement string are considered insignificant unless REPLACE FULL is specified.

EXAMINE statement - EXAMINE DELETE

EXAMINE DELETE finds and deletes part of a string. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #PRODUCT-NAME (A25)
.
END-DEFINE
.
.
#PRODUCT-NAME := 'PAYING TOO MUCH ATTENTION'
.
EXAMINE #PRODUCT-NAME FOR 'TOO MUCH' DELETE
.
```

In this example the resulting value in #STRING would be "PAYING ATTENTION".

If the search string occurs multiple times, by default all occurrences will be deleted. To delete only the first, specify DELETE FIRST.

EXAMINE statement - EXAMINE GIVING LENGTH

EXAMINE GIVING LENGTH returns the length of a string. Consider the following example:

```
EXAMINE #STRING FOR 'E' GIVING LENGTH IN #LENGTH
```

GIVING LENGTH can also be added to the end of an EXAMINE DELETE statement to return the new length. Consider the following example:

```
EXAMINE #STRING FOR 'E' DELETE GIVING LENGTH IN #LENGTH
```

EXAMINE statement - EXAMINE GIVING POSITION

EXAMINE GIVING POSITION finds part of a string and returns a value indicating the position of the first character. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #POSITION (P3)
.
1 #PRODUCT-NAME (A25)
.
END-DEFINE
.
.
#PRODUCT-NAME := ‘PAYING TOO MUCH ATTENTION’
.
EXAMINE #PRODUCT-NAME FOR ‘TOO MUCH’ GIVING POSITION IN #POSITION
.
```

In this example the resulting value in #POSITION would be 8.

EXAMINE statement - EXAMINE GIVING INDEX

EXAMINE GIVING INDEX finds a specified value within an array and returns a value indicating the occurrence containing the specified value. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #INDEX (P2)
.
1 #SHORT-NAME (A3/1:12) CONST<‘JAN’,’FEB’,’MAR’,’APR’,’MAY’,
‘JUN’,’JUL’,’AUG’,’SEP’,’OCT’,’NOV’,’DEC’)
.
END-DEFINE
.
.
EXAMINE #SHORT-NAME(*) FOR ‘MAY’ GIVING INDEX IN #INDEX
.
```

In this example the resulting value in #INDEX would be 5.

The asterisk denotes all occurrences are to be examined. The search can be limited by specifying a range of occurrences. Consider the following example:

```
EXAMINE #SHORT-NAME(7:12) FOR ‘MAY’ GIVING INDEX IN #INDEX
```

If the search is unsuccessful, an index value of zero is returned.

EXAMINE TRANSLATE statement

EXAMINE TRANSLATE converts a string to upper or lower case. Consider the following example:

```
EXAMINE #STRING TRANSLATE INTO UPPER CASE
```

EXAMINE TRANSLATE can also be used to do a character-by-character translation based on the values in an array.

Input and output

Statements in this category control input and output. The following are covered in this section of the course:

- INPUT - INPUT USING MAP
- POS (position)
- REINPUT
- DEFINE WINDOW
- SET WINDOW
- INPUT – INPUT WINDOW USING MAP
- SET KEY
- DISPLAY
- WRITE
- WRITE USING FORM
- FORMAT
- NEWPAGE
- AT TOP OF PAGE
- *PAGE-NUMBER

INPUT statement

The INPUT statement has several uses and takes several forms. In on-line mode it used to obtain data from the user in real-time, most commonly utilizing a pre-defined screen layout that has been stored as a map module.

While the purpose of the INPUT statement is to obtain data, this usually involves prompting, which requires outputting of data. You may find it helpful to think of the INPUT statement as a 2-way interaction with the user, rather than just the human equivalent of reading data from the database. You may also find it helpful to think of a map as a window onto the data within the controlling module, be it a program, subprogram or whatever. Each time you execute an INPUT USING MAP statement, you are giving the user an opportunity to see the data and to manipulate it.

Note that INPUT can also be used to obtain data one field at a time, to dynamically generate a screen layout, to obtain data in batch mode and to obtain data from the NATURAL stack, all of which fall beyond the scope of this course.

INPUT statement - INPUT USING MAP

In on-line applications, INPUT USING MAP is the normal way to display data to the user and to receive input data, using a pre-defined screen layout represented by a map module. Complete examples can be found in the sample application covered in part 2. Detailed information about creating and maintaining maps is contained in appendix A. For now it’s enough to understand the syntax of the INPUT USING MAP statement.

Consider the following examples:

Example 1: INPUT USING MAP

```
DEFINE DATA
LOCAL
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
END-DEFINE
.
.
INPUT
  USING MAP 'IVCLNTM1'
.
```

In this example, map IVCLNTM1 is output without a message and with the cursor positioned at the first input field (by default). Map fields are attached to variables in the program - they are grouped under the dataname #MAP for readability and ease of maintenance.

Example 2: with TEXT option

```
DEFINE DATA
LOCAL
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
1 #MSG-1 (A75) CONST<‘Enter action code and client ID’>
.
END-DEFINE
.
.
INPUT
  WITH TEXT #MSG-1
  USING MAP 'IVCLNTM1'
.
```

In this example, map IVCLNTM1 is output with the contents of #MSG-1 in the message area and the cursor positioned at the first input field.

Example 2: with ALARM option

```
DEFINE DATA
LOCAL
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
1 #MSG (A75)
1 #MSG-1 (A75) CONST<‘Enter action code and client ID’>
1 #MSG-2 (A75) CONST<‘No client with specified ID’>
.
END-DEFINE
.
.
#MSG := #MSG-1
.
INPUT
  WITH TEXT #MSG
  ALARM
  USING MAP 'IVCLNTM1'
.
```

In this example, map IVCLNTM1 is output with the contents of #MSG in the message line and the cursor positioned at the first input field. The alarm (usually a single beep) will sound to attract user attention.

Example 4: with MARK option

```
DEFINE DATA
LOCAL
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
END-DEFINE
.
.
INPUT
  MARK *#MAP.#ACTION
  USING MAP 'IVCLNTM1'
.
```

In this example, map IVCLNTM1 is output with the cursor positioned at the field attached to variable #MAP.#ACTION.

As an alternative, input fields on the map can also be referenced by number (i.e relative position) with 1 representing the first input field, 2 representing the second, and so on. Consider the following example:

```
INPUT
  MARK *1
  USING MAP 'IVCLNTM1'
```

It's good practice to reference map fields by name rather than by number because it allows the sequence of fields to be changed without affecting programming logic.

Example 5: with MARK option using a variable

```
DEFINE DATA
LOCAL
.
1 #FIELD-IN-ERROR (P8)
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
END-DEFINE
.
.
INPUT
  MARK #FIELD-IN-ERROR
  USING MAP 'IVCLNTM1'
.
```

In this example, variable #FIELD-IN-ERROR is used to control cursor positioning. The variable would be initialized before the INPUT statement is executed.

.

Try to avoid multiple INPUT statements. Using variables for messaging (per example 2, above) and cursor positioning (per example 5, above) facilitates this.

System function POS (position)

System function POS (position) returns a number indicating the relative position of a specified map field. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #FIELD-IN-ERROR (P8)
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
END-DEFINE
.
.
#FIELD-IN-ERROR := POS(#MAP.CLIENT-ID)
.
.
```

REINPUT statement

REINPUT passes control to the most recently executed INPUT statement. This provides a useful escape from validation logic when an error is detected. The statement provides a clean way to output an error message and return control to the user.

The INPUT and REINPUT statements must be in the same module and the INPUT statement must not be in a subroutine. Other restrictions apply. The TEXT, MARK and ALARM options specified on the REINPUT statement take precedence over any specified on the original INPUT statement. When input terminates control is passed to the statement following the INPUT, not the statement following the REINPUT.

Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #MAP
  2 #ACTION (A1)
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/1:3)
  2 #AUDIT-TIME (A19)
  2 AUDIT-USER (A8)
  2 AUDIT-PROGRAM (A8)
.
1 #MSG-1 (A75) CONST<‘Enter action code and client ID’>
1 #MSG-2 (A75) CONST<‘No client with specified ID’>
.
END-DEFINE
.
.
INPUT
  WITH TEXT #MSG-1
  MARK *#MAP.#ACTION
  USING MAP 'IVCLNTM1'
.
.
REINPUT FULL
  WITH TEXT #MSG-2
  MARK *#MAP.CLIENT-ID
  ALARM
.
```

I recommend using the FULL option to ensure that all map fields are re-output. It may be a little less efficient but it's a lot less grief for everyone.

Try to avoid multiple REINPUT statements. Using variables for messaging and cursor positioning facilitates this.

DEFINE WINDOW statement

The DEFINE WINDOW statement defines a window which may (or may not) be subsequently activated during processing.

A full-screen window occupies the entire screen. A part-screen window occupies only part of the screen with the original screen as background. When a window is active, only fields within that window are accessible. Although multiple windows can be open simultaneously, only one can be active.

Consider the following examples:

Example 1: Default window

```
DEFINE WINDOW WINDOW-1
```

In this example, WINDOW-1 is defined. Default values will apply as follows:

- window size will be determined automatically based on content
- window position will be determined by the cursor position
- window background will not be reverse video
- window title will be null
- active message area and PF key prompts will be inside the window
- window will be framed

Example 2: Part-screen window

```
DEFINE WINDOW WINDOW-2
  SIZE 14 * 45
  BASE 6/18
  TITLE 'Receivables'
```

In this example, WINDOW-2 is defined. Default values will apply as follows:

- window background will not be reverse video
- active message area will be inside the window
- window will be framed

The window will be 14 lines long and 45 columns wide, including the frame. The top left corner of the frame will be at line 6, column 18. The word "Receivables" will appear centered within the top border of the frame.

Allowing 4 lines total for the top and bottom borders, message area and PF key prompts, plus 5 columns total for the left and right borders and control characters, the usable area within the window is 10 lines of 40 columns.

To provide a consistent user interface, it's good practice to use a consistent window size and position throughout an application, making exceptions on a case by case basis if necessary.

Use example 2 (above) as a starting point for all windows in your next application.

Example 3: Full-screen window

```
DEFINE WINDOW WINDOW-3
  SIZE 24 * 80
  BASE 1/1
  CONTROL SCREEN
  FRAMED OFF
```

In this example, WINDOW-3 is defined. Default values will apply as follows:

- window background will not be reverse video

The window will be 24 lines long and 80 columns wide, starting at line 1, column 1. No frame will be visible. The active message area and PF key prompts will be outside the window (i.e. on the background screen).

Allowing 2 lines total for the message area and PF key prompts, the usable area within the window is 22 lines of 80 columns.

SET WINDOW statement

The SET WINDOW statement either activates (opens) a new window or deactivates (closes) an active one. The window must have been previously defined using a DEFINE WINDOW statement. Consider the following examples:

Example 1: Activating a window

```
SET WINDOW 'WINDOW-2'
```

In this example, WINDOW-2 is opened and activated. The previously active window (if any) is deactivated, but left open.

Example 2: Deactivating a window

```
SET WINDOW OFF
```

In this example, the currently active window is deactivated and closed. The previously active window (if any) is reactivated.

Deactivate the current window before activating a new one, except when a hierarchy of windows is desired and the “stacking” is intentional.

INPUT statement - INPUT WINDOW USING MAP

The WINDOW clause of an INPUT (USING MAP) statement indicates that the map is to be executed for the specified window. The window must have been previously defined using a DEFINE WINDOW statement and optionally activated using a SET WINDOW statement. If the window is too small, data will be truncated at the right and bottom borders.

Consider the following example:

```
DEFINE DATA
.
END-DEFINE
.
DEFINE WINDOW WINDOW-2
  SIZE 14 * 45
  BASE 6/18
  TITLE 'Receivables'
.
.
SET WINDOW 'WINDOW-2'
.
INPUT
  WINDOW='WINDOW-2'
  USING MAP 'IVCLNTM2'
.
SET WINDOW OFF
.
```

In this example, window WINDOW-2 is defined and activated, then map IVCLNTM2 is executed for the window. The window is subsequently deactivated and closed. The underlying screen is unaffected.

To avoid truncation, it's good practice to set the dimensions of the map according to the usable area of the window in which the map will execute. In the above example, the usable area is 10 lines of 40 columns.

Except when using a hierarchy of windows, it is good practice to create a separate subprogram for each window. Any user-entered data can be returned to the calling module via a Parameter Data Area (PDA).

When using a hierarchy of windows, it is good practice to create a single subprogram for each set of windows.

SET KEY statement

The SET KEY statement activates or deactivates PF keys. In on-line programs, the executing module must explicitly activate the keys which the user will need to terminate input. It can also be used to assign names which are then displayed automatically to the user on the PF-key prompt line.

A SET KEY statement should be executed before the first INPUT statement. By default, PF keys are inactive until explicitly activated.

Consider the following examples:

Example 1: Activate PF3

```
SET KEY PF3
```

In this example, PF3 is activated and made available to the user.

Example 2: Deactivate PF7 and PF8

```
SET KEY
  PF7=OFF
  PF8=OFF
```

In this example, PF7 and PF8 are deactivated and made unavailable to the user.

Example 3: Activate PF8 named “Fwd” (forward)

```
SET KEY PF8 NAMED 'Fwd'
```

In this example, PF8 is activated and made available to the user. The PF key prompt line will reflect that PF8 is active and indicate its function.

PF key names are arbitrary, though certain conventions apply. The actual function of PF keys is controlled by program logic. Activating a key simply makes it available to terminate input.

Following an INPUT statement, system variable *PF-KEY can be interrogated to determine how input was terminated, so that the appropriate logic can be executed. This is described in the system variables section later in the course.

DISPLAY statement

A simple way to output information from a module executing in batch mode is to use a DISPLAY statement. Output from DISPLAY statements is organized vertically with column headings, showing the names of the fields being displayed, being generated automatically and output at the top of each page.

Consider the following example, ignoring any unfamiliar syntax:

```
DEFINE DATA
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5)
.
END-DEFINE
.
FOR #DEPT-NBR = 1 TO 5
.
  IF #DOLLARS(#DEPT-NBR) > 0
    DISPLAY #DEPT-NBR #DOLLARS(#DEPT-NBR)
  END-IF
.
END-FOR
.
```

In this example the output might be as follows:

```
          #DEPT-NBR     #DOLLARS
          ---------    -----------
           1               1000.72
           3               2998.05
           4                 63.99
```

Optionally a report number in the range 0 through 31 can be specified. Consider the following example:

```
DISPLAY (1) #DEPT-NBR #DOLLARS(#DEPT-NBR)
```

By using different report numbers, you can generate multiple output streams. When no report number is specified, report number 0 is assumed.

If multiple DISPLAY statements are used for the same report, the automatic column headings are derived from the first one encountered.

I recommend using only one DISPLAY statement per report.

DISPLAY statements are generally used only for casual output, e.g. during debugging. For more formatted output, such as a report, WRITE statements are generally preferred.

WRITE statement

WRITE statements offer more flexibility and control than DISPLAY statements by providing all of the following options:

- Casual output
- Automatic line wrap
- Formatted output using spacers
- Formatted output using tabs
- Formatted output using a combination of spacers and tabs
- WRITE USING FORM

Output from WRITE statements is organized horizontally to facilitate generating reports. WRITE statements are generally placed inside a processing loop and multiple lines of output are generated as a result. By default, a generic heading showing date and time is generated automatically and output at the top of each page.

Consider the following examples, ignoring any unfamiliar syntax:

Example 1: Casual output using WRITE

```
DEFINE DATA
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5)
.
END-DEFINE
.
.
FOR #DEPT-NBR = 1 TO 5
.
  IF #DOLLARS(#DEPT-NBR) > 0
    WRITE
      'Dept' #DEPT-NBR
      'Dollars' #DOLLARS(#DEPT-NBR) (EM=Z,ZZZ,ZZ9.99)
  END-IF
.
END-FOR
.
```

In this example, the output might be as follows:

```
Page      1                                01/01/98  00:00:01

Dept  1 Dollars      1,000.72
Dept  3 Dollars      2,998.05
Dept  4 Dollars         63.99
```

In this example, literal strings are included in the output to make it more understandable. A literal string of '=' can be used to output the actual variable name, which is useful for debugging. Consider the following example:

```
WRITE '=' #DEPT-NBR
```

In the top right corner is the generic heading, which can be suppressed using the NOTITLE option. Consider the following example:

```
WRITE NOTITLE 'Dept' #DEPT-NBR 'Dollars' #DOLLARS(#DEPT-NBR)
```

Example 2: Formatted output with spacers

```
DEFINE DATA
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5)
.
END-DEFINE
.
.
WRITE
  'Dept'
  10X
  'Dollars'
.
FOR #DEPT-NBR = 1 TO 5
.
  IF #DOLLARS(#DEPT-NBR) > 0
    WRITE
      #DEPT-NBR
      7X
      #DOLLARS(#DEPT-NBR) (EM=Z,ZZZ,ZZ9.99)
  END-IF
.
END-FOR
.
```

In this example, the output might be as follows:

```
          Page      1                              01/01/98  00:00:01

          Dept          Dollars
           1           1,000.72
           3           2,998.05
           4              63.99
```

Example 3: Formatted output with tabs

```
DEFINE DATA
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5)
.
END-DEFINE
.
.
WRITE
  'Dept'
  10X
  'Dollars'
.
FOR #DEPT-NBR = 1 TO 5
.
  IF #DOLLARS(#DEPT-NBR) > 0
    WRITE
      1T #DEPT-NBR
      10T #DOLLARS(#DEPT-NBR) (EM=Z,ZZZ,ZZ9.99)
  END-IF
.
END-FOR
.
```

In this example, the output might be as follows:

```
Page      1                              01/01/98  00:00:01

Dept          Dollars
 1           1,000.72
 3           2,998.05
 4              63.99
```

The symbol "/" (slash), other than as part of a literal string, starts a new line. Consider the following example:

```
WRITE 'Dept' #DEPT-NBR / 'Dollars' #DOLLARS(#DEPT-NBR)
```

As with DISPLAY, a report number can optionally be specified. Consider the following example:

```
WRITE (2) 'Dept' #DEPT-NBR / 'Dollars' #DOLLARS(#DEPT-NBR)
```

WRITE USING FORM statement

WRITE USING FORM is another way to generate formatted output. Each different line format is defined by a map module. Each WRITE USING FORM statement generates one or more lines of output depending on the contents of the map module. Consider the following example:

```
WRITE USING FORM 'DEPTTOTS'
```

This statement is covered in more detail in part 3.

FORMAT statement

The FORMAT statement allows you to override the default value of a NATURAL session parameter. It is commonly used to specify line size and/or page size. Line size is the number of characters per line (e.g. 132). Page size is the number of lines per page (e.g. 60). As with WRITE, DISPLAY, etc. a report number can optionally be specified.

Consider the following examples:

```
FORMAT LS=132 PS=60
```

```
FORMAT (2) PS=60
```

FORMAT statements are non-positional and are processed at compile time.

By convention, FORMAT statements are placed before the control logic so that they won't be overlooked by another programmer.

NEWPAGE statement

The NEWPAGE statement allows you to force a new page. As with WRITE, DISPLAY, etc. a report number can optionally be specified.

Consider the following examples:

```
NEWPAGE
```

```
NEWPAGE (2)
```

AT TOP OF PAGE statement

AT TOP OF PAGE allows you to automatically execute one or more statements whenever a new page is started.

The statements to be executed immediately follow the AT TOP OF PAGE and terminate with END-TOPPAGE. It is typically used with logic to print page headings. As with WRITE, DISPLAY, etc. a report number can optionally be specified.

Consider the following example:

```
DEFINE DATA
.
END-DEFINE
.
FORMAT (0) LS=132
AT TOP OF PAGE (0)
  WRITE (0) 57X 'Processing Report'
END-TOPPAGE
.
FORMAT (1) LS=132
AT TOP OF PAGE (1)
  WRITE (1) 60X 'Error Report'
END-TOPPAGE
.
```

In this example, a program writes 2 different reports - the Processing Report is written to report 0 and the Error Report is written to report 1.

When working with multiple reports, it's a good idea to specify the printer number every time - even for printer 0, which is assumed by default. This helps prevent errors caused by omitted printer numbers.

AT TOP OF PAGE statements are non-positional and are processed at compile time.

By convention, AT TOP OF PAGE statements are placed before the control logic so that they won't be overlooked by another programmer.

When working with multiple reports, it's a good idea to place all non-positional statements relating to the same printer together, per the above example.

System variable *PAGE-NUMBER

NATURAL automatically keeps track of page numbers. The current page number for any report can be obtained (in P5 format) using system variable *PAGE-NUMBER.

Consider the following example:

```
DEFINE DATA
.
END-DEFINE
.
FORMAT (0) LS=132
AT TOP OF PAGE (0)
  WRITE (0) 57X 'Processing Report' 23X 'Page' *PAGE-NUMBER(0)
END-TOPPAGE
.
FORMAT (1) LS=132
AT TOP OF PAGE (1)
  WRITE (1) 60X 'Error Report' 24X 'Page' *PAGE-NUMBER(1)
END-TOPPAGE
.
```

Flow control

Statements in this category control logic flow. The following are covered in this section of the course:

- IF
- DECIDE ON
- DECIDE FOR
- FOR
- REPEAT
- PERFORM
- CALLNAT
- FETCH
- STOP

IF statement

IF allows you to conditionally execute one or more statements. The statements to be executed immediately follow the IF and terminate with END-IF. By using an ELSE clause, you can execute statements if the condition is false. Consider the following examples:

Example 1: IF statement

```
DEFINE DATA
LOCAL
.
1 #CORP-NAME (A30)
1 #CORP-NBR (N1)
1 #DAYS (P3)
.
END-DEFINE
.
.
IF #CORP-NBR = 1
  #CORP-NAME := 'Entropy Corporation'
  #DAYS := 30
END-IF
.
```

In this example, the condition is that variable #CORP-NBR contains a value of 1. The relational operator is "=" (equal to).

Relational operators may be used in any conditional statement. The following table details those available.

Relational operators	
Symbol	**Meaning**
= EQ	equal to
NE	not equal to
> GT	greater than
>= GE	greater than or equal to
< LT	less than
<= LE	less than or equal to

Example 2: IF with ELSE

```
DEFINE DATA
LOCAL
.
1 #DAYS (P3)
1 #GOOD-CLIENT (L)
.
END-DEFINE
.
.
IF #GOOD-CLIENT
  #DAYS := 35
ELSE
  #DAYS := 28
END-IF
.
```

In this example, the condition is the value of a logical variable (i.e. TRUE or FALSE).

The following variations in syntax are equally valid:

```
IF #GOOD-CLIENT = TRUE
.
END-IF
```

```
IF #GOOD-CLIENT = FALSE
.
END-IF
```

```
IF NOT #GOOD-CLIENT
.
END-IF
```

NOT is a boolean operator, as explained below.

Example 3: Nested IF with ELSE

```
DEFINE DATA
LOCAL
.
1 #CORP-NAME (A30)
1 #CORP-NBR (N1)
.
END-DEFINE
.
.
IF #CORP-NBR = 1
  #CORP-NAME := 'Northern Enterprises'
ELSE
  IF #CORP-NBR = 2
    #CORP-NAME := 'Southern Hospitality Corp'
  ELSE
    IF #CORP-NBR = 3
      #CORP-NAME := 'Eastern Partnership'
    ELSE
      #CORP-NAME := 'Western Vista'
    END-IF
  END-IF
END-IF
.
```

Although this example contains valid syntax, in general, nested IF statements are avoided in favor of DECIDE statements (see below).

For nested IF statements such as this, correct and consistent indentation greatly improves readability.

Example 4: IF with compound conditions

```
DEFINE DATA
LOCAL
.
1 #CORP-NBR (N1)
1 #DAYS (P3)
1 #GOOD-CLIENT (L)
.
END-DEFINE
.
.
IF (#CORP-NBR = 1) AND (#GOOD-CLIENT = TRUE)
  #DAYS := 35
ELSE
  #DAYS := 28
END-IF
.
```

You can specify compound and/or negative conditions using “boolean operators”, which are as follows:

Boolean operators	
Symbol	**Meaning**
AND	the conditions separated by AND are both true
OR	at least one of the conditions separated by OR is true
NOT	the condition is false

I recommend using parentheses in compound condition statements, per the above example, to avoid errors and to improve readability.

DECIDE ON statement

DECIDE ON allows you to list possible values for a field and to specify different actions for each value, in a very readable way. DECIDE ON statements terminate with END-DECIDE. Consider the following examples:

Example 1: DECIDE ON

```
DEFINE DATA
LOCAL
.
1 #CORP-NAME (A30)
1 #CORP-NBR (N1)
.
END-DEFINE
.
.
DECIDE ON FIRST VALUE OF #CORP-NBR
  VALUE 1
    #CORP-NAME := 'Northern Enterprises'
  VALUE 2
    #CORP-NAME := 'Southern Hospitality Corp'
  VALUE 3
    #CORP-NAME := 'Eastern Partnership'
  NONE
    #CORP-NAME := 'Western Vista'
END-DECIDE
.
```

This example is a clean alternative to the nested IF ELSE logic shown earlier.

Multiple values can be specified in a single VALUE clause, for example:

```
VALUE 1, 5, 7:10
```

A NONE clause is mandatory to specify an action for an undocumented value.

Example 2: DECIDE ON with IGNORE statement

```
DEFINE DATA
LOCAL
.
1 #CORP-NAME (A30)
1 #CORP-NBR (N1)
.
END-DEFINE
.
.
DECIDE ON FIRST VALUE OF #CORP-NBR
  VALUE 1
    #CORP-NAME := 'Northern Enterprises'
  VALUE 2
    #CORP-NAME := 'Southern Hospitality Corp'
  VALUE 3
    #CORP-NAME := 'Eastern Partnership'
  VALUE 4
    #CORP-NAME := 'Western Vista'
  NONE
    IGNORE
END-DECIDE
.
```

Since a NONE clause is mandatory, it is sometimes necessary to specify a "null" action. This is done using an IGNORE statement.

IGNORE can also be used in conjunction with other statements, but is most commonly used with DECIDE and IF.

DECIDE FOR statement

DECIDE FOR allows you to list possible conditions and to specify different actions for each condition. DECIDE FOR statements terminate with END-DECIDE. Consider the following examples:

Example 1: DECIDE FOR FIRST

```
DEFINE DATA
LOCAL
.
1 #CORP-NAME (A30)
1 #CORP-NBR (N1)
.
END-DEFINE
.
DECIDE FOR FIRST CONDITION
  WHEN #CORP-NBR = 1
    #DAYS := 28
  WHEN #GOOD-CLIENT = TRUE
    #DAYS := 35
  WHEN NONE
    #DAYS := 30
END-DECIDE
.
```

Unlike a DECIDE ON statement where one field is referenced and the actions vary by the value of the field, a DECIDE FOR statement is much more flexible. Each condition within the DECIDE can reference any field.

In a DECIDE FOR statement the sequence of conditions is very significant. Conditions are evaluated sequentially, i.e. from the top down.

In this example, when any condition is satisfied, the specified actions will be taken and control will pass the statement after END-DECIDE.

In this example, #CORP-NBR takes precedence over #GOOD-CLIENT because of the sequence in which the conditions are coded. Any Northern Enterprises client (good or bad) gets only 28 days to pay. Good clients of Southern Hospitality Corp, Eastern Partnership and Western Vista get a full 35 days to pay, and even bad clients get 30.

The WHEN NONE clause is mandatory.

Example 2: DECIDE FOR EVERY

```
DEFINE DATA
LOCAL
.
1 #CORP-NAME (A30)
1 #CORP-NBR (N1)
.
END-DEFINE
.
.
DECIDE FOR EVERY CONDITION
  WHEN #CORP-NBR = 1
    #DAYS := 28
  WHEN #GOOD-CLIENT = TRUE
    #DAYS := 35
  WHEN NONE
    #DAYS := 30
END-DECIDE
.
```

In this example, when a condition is satisfied, the specified actions will be taken and additionally every subsequent condition will be evaluated until END-DECIDE is reached.

I have deliberately kept this example the same as the previous one (except changing FIRST to EVERY) to make a point - that the result will be entirely different. Good clients of Northern Enterprises would get 35 days to pay (instead of 28 as in the previous example). Bad clients of Northern Enterprises would continue to get 28 days (because the NONE condition wouldn't be satisfied). A better, although contrived, example of DECIDE FOR EVERY follows.

Example 3: DECIDE FOR EVERY with ANY and ALL clauses

```
DEFINE DATA
LOCAL
.
1 #BLUE (L) INIT<TRUE>
1 #RED (L) INIT<FALSE>
1 #YELLOW (L) INIT<TRUE>
.
END-DEFINE
.
DECIDE FOR EVERY CONDITION
  WHEN #BLUE
    WRITE 'We have blue paint'
  WHEN #RED
    WRITE 'We have red paint'
  WHEN #YELLOW
    WRITE 'We have yellow paint'
  WHEN #BLUE AND #RED
    WRITE 'We can make purple'
  WHEN #BLUE AND #YELLOW
    WRITE 'We can make green'
  WHEN #RED AND #YELLOW
    WRITE 'We can make orange'
  WHEN ANY
    WRITE 'Yes, we have paint'
  WHEN ALL
    WRITE 'We can make brown'
  WHEN NONE
    WRITE 'No, we don't have paint'
END-DECIDE
.
```

Optionally, a WHEN ANY clause can be placed before the mandatory WHEN NONE clause. If present, the specified actions will be taken if any of the preceding conditions are true.

Similarly, a WHEN ALL clause can optionally be placed before the mandatory WHEN NONE clause. If present, the specified actions will be taken if all of the preceding conditions are true.

In this example, the following messages would be written:

```
We have blue paint
We have yellow paint
We can make green
Yes, we have paint
```

FOR statement

FOR automatically increments (or decrements) a control variable which can be used by logic contained inside the associated processing loop. The top of the loop is marked by the FOR statement, the bottom is marked by END-FOR.

Consider the following examples:

Example 1: FOR loop

```
DEFINE DATA
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5) /* 5 DEPARTMENTS
.
END-DEFINE
.
.
FOR #DEPT-NBR = 1 TO 5
  IF #DOLLARS(#DEPT-NBR) > 0
.
  END-IF
END-FOR
.
```

In this example, a loop is executed 5 times - once for each department. During each iteration, the control variable #DEPT-NBR references one of the 5 departments.

During execution of a FOR loop, the control variable should not be manipulated. On exit from the loop a value should not be assumed for the control variable.

By default the control variable is incremented by 1. It can be incremented by a different number, or decremented: Consider the following examples:

```
FOR #ODD-INDEX = 1 TO 99 STEP 2
```

```
FOR #DOWN-INDEX = 99 TO 1 STEP -1
```

Example 2: Nested FOR loop

```
DEFINE DATA
.
1 #DEPT-NBR (P1)
1 #DOLLARS(P7.2/1:5,1:12) /* 5 DEPARTMENTS, 12 MONTHS
1 #MONTH-NBR (P2)
.
END-DEFINE
.
.
FOR #DEPT-NBR = 1 TO 5
  FOR #MONTH-NBR = 1 TO 12
    IF #DOLLARS(#DEPT-NBR,#MONTH-NBR) > 0
.
    END-IF
  END-FOR
END-FOR
.
```

In this example there are 2 FOR loops, one nested inside the other. The outermost loop is executed 5 times - once for each department. The innermost loop is executed 12 times - once for each month. During each iteration, the control variable #DEPT-NBR references one of the 5 departments and control variable #MONTH-NBR references one of the 12 months.

In a hierarchy of processing loops, each loop executes completely per single iteration of the surrounding loop.

REPEAT statement

REPEAT allows you to create a processing loop around logic so that it can be executed multiple times until a specified condition becomes true. The top of the loop is marked by the REPEAT statement, the bottom is marked by END-REPEAT.

Consider the following example:

```
DEFINE DATA
.
1 #DOLLARS (P7.2/1:5)
1 #INDEX (P3)
1 #MAX-INDEX (P3)
.
END-DEFINE
.
.
RESET #INDEX
.
REPEAT UNTIL #INDEX GE #MAX-INDEX
  ADD 1 TO #INDEX
  IF #DOLLARS(#INDEX) > 0
    WRITE ‘=‘ #DOLLARS(#INDEX)
  END-IF
END-REPEAT
.
```

The UNTIL clause is optional. Without one, an unconditional loop is created. Unconditional loops can be terminated using an ESCAPE statement, as described in a later section.

Generally, REPEAT is used when the number of iterations is not fixed, and termination of the processing loop is dependent on a dynamic condition.

Generally, FOR is used when the number of iterations is fixed. In this scenario, using a FOR statement rather than a REPEAT statement adds readability.

PERFORM statement

PERFORM executes a subroutine. If a subroutine with the specified name is contained within the same module as the PERFORM statement, it will be executed. If not, NATURAL will look for an external subroutine. An execution time error will occur if the subroutine is not found. When the subroutine finishes executing, control passes to the statement following the PERFORM.

Consider the following example:

```
DEFINE DATA
.
END-DEFINE
.
.
PERFORM CALC-AMOUNTS-DUE
.
.
DEFINE SUBROUTINE CALC-AMOUNTS-DUE
************************************************************
** Calculates amount(s) due.
************************************************************
.
END-SUBROUTINE /* CALC-AMOUNTS-DUE
.
```

CALLNAT statement

CALLNAT executes a subprogram. An execution time error will occur if the subprogram is not found. When the subprogram finishes executing, control passes to the statement following the CALLNAT.

Consider the following example:

```
DEFINE DATA
LOCAL USING IVINSTA1 /* PDA FOR SUBPROGRAM IVINSTN1
LOCAL
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
  2 PRODUCT-ID
  2 QUANTITY
  2 TOTAL-PRICE
  2 PAID-AMOUNT
  2 C*BILL-DATE
.
END-DEFINE
.
.
IVINSTA1.TOTAL-PRICE := RECEIVABLE.TOTAL-PRICE
IVINSTA1.NBR-INSTALLMENTS := RECEIVABLE.C*BILL-DATE
CALLNAT 'IVINSTN1' IVINSTA1
.
```

FETCH statement

The FETCH statement allows you to to invoke and pass control to a NATURAL program. It can be used in a program or at a higher level, i.e. in a subprogram or subroutine. It has 2 forms, as follows:

- FETCH
- FETCH RETURN

FETCH statement – FETCH

FETCH causes the currently active module and all lower level modules to terminate and control to pass to the "fetched" program. Consider the following example:

```
FETCH 'IVEXTRP1'
```

The sequence in which batch programs execute is best controlled by JCL statements. The only reason to put this kind of logic in NATURAL programs would be to overcome the limitations of JCL (for example, in a situation where the sequence of programs was very dynamic, requiring decisions to be made based on the incoming data).

FETCH statement – FETCH RETURN

FETCH RETURN causes the currently active module and all lower level modules to be suspended (not terminated) and control to pass temporarily to the "fetched" program. When the "fetched" program terminates, control returns to the "fetching" module. Effectively, FETCH RETURN allows you to execute a program as though it were a subprogram or subroutine.

Consider the following example:

```
FETCH RETURN 'IVEXTRP1'
```

Generally, any logic which needs to be executed as a subprogram should be coded as such. FETCH RETURN is used only in rare circumstances.

STOP statement

STOP causes application execution to cease. Consider the following example:

```
STOP
```

A STOP statement can be used in a program or at a higher level, i.e. in a subprogram or subroutine. Regardless of where it is used, execution of the active module and all lower level modules ceases. STOP is generally used only to terminate an application following a fatal error.

Database access

Statements in this category allow you to retrieve and maintain records in the database. There are some related system variables to facilitate this. The following are covered in this section of the course:

- READ
- Labels
- *ISN
- FIND
- *COUNTER
- GET
- STORE
- UPDATE
- DELETE

There are 3 statements for retrieving records, as follows:

Database retrieval statements	
Statement	**Use**
READ	to access one or multiple records with different keys even if elements of the key are unknown
FIND	to access one or multiple records using search criteria
GET	to access a single record when the ISN is known

READ statement

A READ statement has a processing loop associated with it. READ defines the top of the loop. END-READ defines the bottom of the loop. It has multiple forms as described in the following table.

READ statement	
Statement	**Use**
READ PHYSICAL (most efficient)	when all or most of the records in the file are to be presented when the sequence of records is not important when there is no risk of increasing record size during update
READ BY ISN	when some or all of the records in the file are to be presented when the sequence of records is not important (or ISN sequence is required) when there is a risk of increasing record size during update
READ LOGICAL (least efficient)	when some or all of the records in the file are to be presented when only records within a specific key range are to be presented when the sequence of records presented is important

READ statement – READ PHYSICAL

READ PHYSICAL presents records in physical (usually random) sequence. Consider the following example:

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  READ-RECEIVABLE.
READ RECEIVABLE IN PHYSICAL SEQUENCE
.
  WRITE
    '=' *ISN(READ-RECEIVABLE.)
    '=' RECEIVABLE.CLIENT-ID
.
END-READ
.
```

In this example every record in the file would be presented. The output might be as follows:

```
          ISN: 0000000003 CLIENT-ID: 10060611
          ISN: 0000000001 CLIENT-ID: 10096786
          ISN: 0000000002 CLIENT-ID: 10078240
          etc.
```

Labels

Any NATURAL statement which initiates a processing loop can optionally be given a label. In the above example, the READ statement is labeled *READ-RECEIVABLE*. Labels are simply names which end with a period. They provide reference points to improve readability and in some cases actually affect the code generated by NATURAL.

System variable *ISN

System variable *ISN is used to obtain the ISN of a previously accessed record. It is good practice to use this variable in conjunction with a label, as illustrated in the example which follows.

READ statement – READ BY ISN

Presents records in ISN sequence. Optionally, you can specify a starting and/or ending ISN. Consider the following example:

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  READ-RECEIVABLE.
READ RECEIVABLE BY ISN
.
  WRITE
    '=' *ISN(READ-RECEIVABLE.)
    '=' RECEIVABLE.CLIENT-ID
.
END-READ
.
```

In this example every record in the file would be presented. The output might be as follows:

```
        ISN: 0000000001 CLIENT-ID: 10096786
        ISN: 0000000002 CLIENT-ID: 10078240
        ISN: 0000000003 CLIENT-ID: 10060611
        etc.
```

READ statement – READ LOGICAL

Presents records in logical (key) sequence. Optionally, you can specify a starting and/or ending key value. Consider the following examples:

Example 1: READ LOGICAL

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  READ-RECEIVABLE.
READ RECEIVABLE BY CLIENT-ID
.
  WRITE
    '=' *ISN(READ-RECEIVABLE.)
    '=' RECEIVABLE.CLIENT-ID
.
END-READ
.
```

In this example every record in the file would be presented. The output might be as follows:

```
        ISN: 0000000153 CLIENT-ID: 10000051
        ISN: 0000030761 CLIENT-ID: 10000309
        ISN: 0000000082 CLIENT-ID: 10002448
        etc.
```

Example 2: READ LOGICAL with range

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  READ-RECEIVABLE.
READ RECEIVABLE BY CLIENT-ID FROM 10000000 THRU 10000999
.
  WRITE
    '=' *ISN(READ-RECEIVABLE.)
    '=' RECEIVABLE.CLIENT-ID
.
END-READ
.
```

In this example only records with key values in the specified range would be presented. The output might be as follows:

```
ISN: 0000000153 CLIENT-ID: 10000051
ISN: 0000030761 CLIENT-ID: 10000309
etc.
```

The FROM and THRU clauses can be used independently. A FROM clause can be used with any type of key. A THRU clause cannot be used with a multi-element key.

The starting and ending values specified do not need to be exact, i.e. records with the specified values do not have to exist. FROM will start at the first record with the specified value or the next highest in-range value. THRU will stop at the last record with the specified value or the highest in-range value.

There are several syntactical variations for defining a range of records, but FROM and THRU are the simplest.

READ LOGICAL is the most commonly used form of the READ statement. A FROM clause is commonly used.

READ BY ISN is not commonly used.

With all forms of the READ statement you can optionally specify a limit for the number of records to be presented. Consider the following examples:

```
READ (1) RECEIVABLE BY ISN FROM #ISN
```

In this example, a maximum of 1 record would be presented.

```
READ (100) RECEIVABLE BY CLIENT-ID
```

In this example, a maximum of 100 records would be presented.

```
READ (01000) RECEIVABLE IN PHYSICAL SEQUENCE
```

In this example, a maximum of 1,000 record would be presented. Use of a leading zero avoids a 4 digit number which would be interpreted as a reference to another statement and reported as a syntax error. As you will see in part 3, NATURAL assigns a 4-digit line number to every line of source. The line numbers can be used instead of labels, but I recommend against this practice.

FIND statement

The FIND statement has several forms. The essential ones are as follows:

- FIND
- FIND NUMBER

FIND and FIND NUMBER are completely different - FIND retrieves data whereas FIND NUMBER queries the database. FIND NUMBER is described later under the heading “Database Query”.

FIND statement - FIND

A FIND statement allows you to search for records using potentially complex criteria. Internally, this is managed by building a list of ISNs representing records that satisfy the criteria. Only after the entire list is complete, is the first record presented for processing. By default, records are presented in ISN sequence. Since the ISN list is pre-built, any external updates that occur during execution of a FIND statement do not affect a record's eligibility for processing in that FIND loop.

The FIND statement provides the capability to save the ISN list it builds. Although, there are situations where this is necessary, they are rare.

Since complex searches are seldom needed and FIND statement syntax can be difficult and the statement is enormously inefficient when used improperly, complex FIND statement are generally avoided. For these reasons, only the basic use of FIND is covered in this course.

A FIND statement has a processing loop associated with it. FIND defines the top of the loop. END-FIND defines the bottom of the loop.

Consider the following example:

```
DEFINE DATA
.
1 #CLIENT-ID (N8)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  FIND-RECEIVABLE.
FIND RECEIVABLE WITH CLIENT-ID = #CLIENT-ID
  IF NO RECORDS FOUND
    WRITE 'NO RECEIVABLES FOR CLIENT' #CLIENT-ID
  END-NOREC
.
END-FIND
.
```

In this example only records for the specified client would be presented.

The NO RECORDS logic in this example is optional. If coded, NO RECORDS logic is executed only when no records are presented. It is non-positional, so can be placed anywhere within the FIND loop.

By convention, the NO RECORDS clause is placed immediately after the FIND statement so that it won't be overlooked by another programmer.

As with the READ statement you can optionally specify a limit for the number of records to be presented. Consider the following example:

```
FIND (1) RECEIVABLE WITH CLIENT-ID = #CLIENT-ID
```

System variable *COUNTER

System variable *COUNTER can be used within a READ or FIND loop to determine the iteration number. The returned value is in P10 format. Consider the following example:

```
DEFINE DATA
.
1 #CLIENT-ID (N8)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  FIND-RECEIVABLE.
FIND RECEIVABLE WITH CLIENT-ID = #CLIENT-ID
.
  IF *COUNTER(FIND-RECEIVABLE.) = 1 /* first record found
.
.
END-FIND
.
```

In this example, system variable *COUNTER is used to determine how many records have been presented for a specific client. Different actions are taken when the first record is presented.

Using *COUNTER in conjunction with a label, as illustrated in the above example, is recommended.

GET statement

A GET statement presents a single record for processing when the ISN is known. Unlike READ and FIND, there is no associated processing loop. Consider the following example:

```
DEFINE DATA
.
1 #CLIENT-ID (N8)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  FIND-RECEIVABLE.
FIND RECEIVABLE WITH CLIENT-ID = #CLIENT-ID
.
    GET-RECEIVABLE.
  GET RECEIVABLE *ISN(FIND-RECEIVABLE.)
.
END-FIND
.
.
```

Executing a GET statement with an invalid (or unassigned) ISN causes an execution time error. For this reason, some programmers prefer to code READ (1) BY ISN and explicitly check that the correct record was read. This is less efficient, but much safer when ISNs are stored on a file - a practice that is not recommended except in special situations.

STORE statement

STORE adds a record to a database file. Data fields must be populated before the STORE is executed. Consider the following example ignoring any unfamiliar syntax:

```
DEFINE DATA
LOCAL
.
1 #CLIENT-ID (N8)
1 #ISN (P10)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
  2 UNPAID-FLAG
  2 PRODUCT-ID
  2 QUANTITY
  2 TOTAL-PRICE
  2 PAID-AMOUNT
  2 BILL-DATE (1:12)
  2 AUDIT-TIME
  2 AUDIT-USER
  2 AUDIT-PROGRAM
  2 AUDIT-COUNTER
.
END-DEFINE
.
.
RESET RECEIVABLE
RECEIVABLE.CLIENT-ID := #CLIENT-ID
.
RECEIVABLE.AUDIT-TIME := *TIMX
RECEIVABLE.AUDIT-PROGRAM := *PROGRAM
RECEIVABLE.AUDIT-USER := *USER
RECEIVABLE.AUDIT-COUNTER := 1
.
  STORE-RECEIVABLE.
STORE RECEIVABLE
.
#ISN := *ISN(STORE-RECEIVABLE.)
.
```

In this example the ISN of the newly created record is saved in variable #ISN for future use.

Generally ISNs are obtained by successfully executing a STORE, READ or FIND statement, as illustrated in the preceding examples. In modules containing multiple STORE, READ and/or FIND statements, use labels to improve readability and avoid errors.

Records are not irrevocably added to the database until all pending database updates are committed. This is covered in detail in the next section, entitled Transaction Control Statements.

UPDATE statement

UPDATE overlays an existing database record. The record must be brought in (using READ, FIND or GET) before it can be changed and written back to the database. Consider the following example:

```
DEFINE DATA
.
1 #INSTALLMENT-NBR (P3)
1 #ISN (P10)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
    GET-RECEIVABLE.
  GET RECEIVABLE #ISN
.
  ADD 30 TO RECEIVABLE.BILL-DATE(#INSTALLMENT-NBR)
  RECEIVABLE.AUDIT-TIME := *TIMX
  RECEIVABLE.AUDIT-PROGRAM := *PROGRAM
  RECEIVABLE.AUDIT-USER := *USER
  ADD 1 TO RECEIVABLE.AUDIT-COUNTER
*
  UPDATE (GET-RECEIVABLE.)
.
.
```

In this example, the record is brought in using a GET statement labeled *GET-RECEIVABLE* which the UPDATE statement refers back to.

Unless you're updating all the records presented in a READ or FIND loop, it's good practice to use GET to bring in records for update. This avoids "holding" records unnecessarily.

It's important to update audit trail fields every time a record is changed, no matter how trivial the change may be.

Records are not irrevocably updated until all pending database updates are committed. This is covered in detail in the next section, entitled Transaction Control Statements.

DELETE statement

DELETE purges a record from a database file. The record must be brought in (using READ, FIND or GET) before it can be deleted. Consider the following example:

```
DEFINE DATA
.
1 #ISN (P10)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  GET-RECEIVABLE.
GET RECEIVABLE #ISN
.
DELETE (GET-RECEIVABLE.)
.
```

In this example, the record is brought in using a GET statement labeled *GET-RECEIVABLE* which the DELETE statement refers back to.

Unless you're deleting all the records presented in a READ or FIND loop, it's good practice to use GET to bring in records for deletion. This avoids "holding" records unnecessarily.

Records are not irrevocably deleted until all pending database updates are committed. This is covered in detail in the next section, entitled Transaction Control Statements.

Transaction control

Statements in this category allow you to commit database updates, back-out database updates, and retrieve previously committed restart data. The following are covered in this section of the course:

- Logical transactions
- BACKOUT TRANSACTION
- END TRANSACTION
- GET TRANSACTION DATA

Logical Transactions

All updates that occur between execution of either an END TRANSACTION or BACKOUT TRANSACTION statement constitute a logical transaction. It may help to think of a logical transaction as a unit of work. All update processes are divided into logical transactions. For example in an on-line application, all updates that are done as a result of 1 user input might constitute a logical transaction. In a batch application where the input stream is sorted into client sequence, processing all the input records for a client might constitute a logical transaction.

BACKOUT TRANSACTION statement

BACKOUT TRANSACTION backs out all updates done since the last-executed END TRANSACTION or BACKOUT TRANSACTION statement. Consider the following example:

```
BACKOUT TRANSACTION
```

END TRANSACTION statement

END TRANSACTION commits all updates done since the last-executed END TRANSACTION or BACKOUT TRANSACTION statement. Consider the following example:

```
END TRANSACTION
```

Optionally restart data can be saved using an END TRANSACTION statement and subsequently retrieved when the program is re-executed. This is covered in the next example.

It's considered bad practice to code BACKOUT TRANSACTION and END TRANSACTION statements within subprograms, subroutines or helproutines. In a well-designed application, each subprogram, subroutine and helproutine contains logic for a specialized and usually very limited function. The decision to commit or backout database updates belongs at a higher level - when all of the different specialized functions which contribute to a "unit of work" have been executed. There are exceptions, but think twice before coding a BACKOUT TRANSACTION or END TRANSACTION statement in any type of module other than a program.

GET TRANSACTION DATA statement

GET TRANSACTION DATA retrieves restart data previously committed with an END TRANSACTION statement, if any. Consider the following example.

In this example, a batch program reads an entire database file in client ID sequence and accumulates statistics which are summarized on another file.

```
DEFINE DATA
.
1 #RESTART-DATA (A100)
1 REDEFINE #RESTART-DATA
  2 CLIENT-ID (N8)
.
.
1 CLIENT VIEW OF CLIENT
  2 CLIENT-ID
.
.
END-DEFINE
.
GET TRANSACTION DATA #RESTART-DATA
.
  READ-CLIENT.
READ CLIENT BY CLIENT-ID FROM #RESTART-DATA.CLIENT-ID
.
  ADD 1 TO #ET-COUNT
  IF #ET-COUNT GE 200
    END TRANSACTION #RESTART-DATA
    RESET #ET-COUNT
  END-IF
.
  IF CLIENT.CLIENT-ID > #RESTART-DATA.CLIENT-ID
    #RESTART-DATA.CLIENT-ID := CLIENT.CLIENT-ID
.
  END-IF
END-READ
.
RESET #RESTART-DATA
END TRANSACTION #RESTART-DATA
.
```

This program retrieves any restart data leftover from a prior run using a GET TRANSACTION DATA statement.

File CLIENT is accessed using a READ LOGICAL statement with a FROM clause. The starting key value is derived from the restart data.

For every 200 records read (an arbitrary number), updates to the other files (not shown) are committed using an END TRANSACTION statement. At the same time, the CLIENT-ID for the last client processed is saved as restart data.

Note that the END TRANSACTION statement is deliberately placed before the logic to determine whether the restart point has been reached. If the restart point has not been reached, the restart data will simply be re-saved without having been changed. This may seem redundant, but the reason for coding it this way will be explained later.

The final END TRANSACTION statement will commit any remaining updates and erase the restart data. Next time the program is executed, it will start from the first client.

There are no rules for the format of restart data since the information that needs to be saved will vary from application to application. Generally the information saved includes the restart point and any accumulated totals which would otherwise be lost. Restart logic is revisited in part 3 of the course.

The functionality to support restart data requires databases to be set-up in a standard way. Check with your Database Administrator, or write a test program, before assuming a standard set-up.

Database query

Statements in this category allow you to query the database without accessing records. The following are covered in this section of the course:

- FIND NUMBER
- HISTOGRAM
- *NUMBER

FIND statement – FIND NUMBER

FIND NUMBER is a variation of the FIND statement described earlier. It returns the number of records with a specific key value. The resulting number is placed in system variable *NUMBER. This use of FIND does not create a processing loop. Consider the following example:

```
DEFINE DATA
.
1 #CLIENT-ID (N8)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  NBR-RECEIVABLE.
FIND NUMBER RECEIVABLE WITH CLIENT-ID = #CLIENT-ID
.
IF *NUMBER(NBR-RECEIVABLE.) = 0
  WRITE 'NO RECEIVABLES FOR CLIENT' #CLIENT-ID
END-IF
.
```

In this example, the FIND NUMBER statement is labeled *NBR-RECEIVABLE* which system variable *NUMBER refers back to.

Since no data records are accessed, this statement is very efficient.

HISTOGRAM statement

The HISTOGRAM statement allows you to determine what unique key values exist on a file. A HISTOGRAM statement has a processing loop associated with it. HISTOGRAM defines the top of the loop. END-HISTOGRAM defines the bottom of the loop. Optionally, you can specify a starting and/or ending key value. Consider the following examples:

Example 1: HISTOGRAM

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
END-DEFINE
.
.
  HIST-RECEIVABLE.
HISTOGRAM RECEIVABLE FOR CLIENT-ID
.
  WRITE '=' RECEIVABLE.CLIENT-ID
.
END-HISTOGRAM
.
```

In this example the id of every client with receivables would be presented. There would be one line of output for each.

The record definition used in a HISTOGRAM statement must only define the relevant key field. Since data records are not accessed, there is no reason to define other fields.

Although the above rule applies to all HISTOGRAM statements, including a HISTOGRAM of a multi-element key, you can redefine a multi-element key into its component parts and obtain some data that way. Consider the following example:

Example 2: HISTOGRAM for multi-element key

For this example, assume that database file RECEIVABLES is defined with CLIENT-ID as a single element key plus CLIENT-PRODUCT as a multi-element key (a superdescriptor in ADABAS terminology).

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-PRODUCT
  2 REDEFINE CLIENT-PRODUCT
    3 #CLIENT-ID (N8)
    3 #PRODUCT-ID (N6)
.
END-DEFINE
.
.
  HIST-RECEIVABLE.
HISTOGRAM RECEIVABLE FOR CLIENT-PRODUCT
.
  WRITE
    '=' RECEIVABLE.#CLIENT-ID
    '=' RECEIVABLE.#PRODUCT-ID
.
END-HISTOGRAM
.
```

In this example every unique combination of client id and product id would be presented. There would be one line of output for each, showing the individual elements.

Since the record definition used in a HISTOGRAM statement must not reference fields other than the relevant key field, the names CLIENT-ID and PRODUCT-ID cannot be used. To get around this the names are prefixed with '#'.

Because we have a suitable key, using HISTOGRAM with REDEFINE allows us to find out the products for which a client has receivables in a very efficient way, without actually "reading" any data records.

System variable *NUMBER

As mentioned, system variable *NUMBER is used to obtain the result from a FIND NUMBER statement. It can also be used with a HISTOGRAM statement to determine the number of records with a particular key value. Consider the following example:

For this example, assume that database file RECEIVABLES is defined with CLIENT-ID as a single element key plus CLIENT-PRODUCT as a multi-element key (a superdescriptor in ADABAS terminology).

Further assume that a client may have multiple orders and multiple receivables for the same product.

```
DEFINE DATA
.
1 #TOT-CLIENTS-8101 (P5)
1 #TOT-RECEIVABLES-8101 (P5)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-PRODUCT
  2 REDEFINE CLIENT-PRODUCT
    3 #CLIENT-ID (N8)
    3 #PRODUCT-ID (N6)
.
END-DEFINE
.
.
  HIST-RECEIVABLE.
HISTOGRAM RECEIVABLE FOR CLIENT-PRODUCT
.
  IF RECEIVABLE.#PRODUCT-ID = 8101
    ADD *NUMBER(HIST-RECEIVABLE.) TO #TOT-RECEIVABLES-8101
    ADD 1 TO #TOT-CLIENTS-8101
  END-IF
.
END-HISTOGRAM
.
```

In this example, the HISTOGRAM statement is labeled *HIST-RECEIVABLE* which system variable *NUMBER refers back to.

In this example, the total number of receivables for product 8101 is accumulated in variable #TOT-RECEIVABLES-8101. The total number of clients with receivables for product 8101 is accumulated in variable #TOT-CLIENTS-8101.

Sequential files

Statements in this category allow you to read and write files, commonly known as “flat files”, that are not part of the database and can only be accessed sequentially. The following statements are covered in this section of the course:

- READ WORK FILE
- WRITE WORK FILE

Sequential files do not need to be explicitly opened or closed.

READ WORK FILE statement

This statement allows you to read a sequential file. A READ WORK FILE statement has a processing loop associated with it. READ WORK FILE defines the top of the loop. END-WORK defines the bottom of the loop. Consider the following example:

```
DEFINE DATA
.
1 INVOICE
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/3)
  2 PRODUCT-ID (N6)
  2 PRODUCT-NAME (A20)
.
END-DEFINE
.
.
READ WORK FILE 1 INVOICE
.
  NEWPAGE
.
  WRITE NOTITLE USING FORM 'IVPRNTM1'
.
  WRITE NOTITLE USING FORM 'IVPRNTM2'
.
  WRITE NOTITLE USING FORM 'IVPRNTM3'
.
END-WORK
.
```

In this example, output is generated using WRITE USING FORM statements. There would be a whole page of output for each input record.

Work file numbering is arbitrary, but a common practice is to give input files lower numbers than output files, and to start the numbering from 1.

The arbitrary work file numbers are translated into dataset names using JCL. For this reason, programs which read and/or write work files can only be executed in batch mode.

There must be data definition statements to define the incoming data. Since the input data is organized into records, it is good practice to define it using a record structure and reference it by the group name, as illustrated in the above example.

If multiple modules use the same sequential file (e.g. one module to write it, another to read it), it's good practice to define the record using a local data area (LDA) which can be shared by all. This eliminates the risk of problems caused by inconsistent data definitions.

I recently worked on a conversion project that required me to identify all system interfaces. Since sequential files are often used to interface batch systems, having such files defined in one LDA in a common library, rather than multiple modules in different libraries, greatly simplifies this type of research.

There's one common scenario where using a record structure and/or an LDA doesn't make sense. This is when a program needs access to only one or two items of data (e.g. some kind of control parameter, such as a date) which can be coded into the JCL. In this scenario, the data should be read directly into the field(s), for example:

```
READ WORK FILE 1
  #START-DATE
  #UPDATE-MODE
```

READ WORK FILE ONCE is a variation that allows you to read a sequential file one record at a time, without creating a processing loop. Since it's rarely used, it is outside the scope of this course.

WRITE WORK FILE statement

This statement allows you to write to a sequential file. Consider the following example:

```
DEFINE DATA
.
1 INVOICE
  2 CLIENT-ID (N8)
  2 BILLING-NAME (A40)
  2 BILLING-ADDRESS (A40/3)
  2 PRODUCT-ID (N6)
  2 PRODUCT-NAME (A20)
  2 QUANTITY (N3)
  2 TOTAL-PRICE (N7.2)
.
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
  2 UNPAID-FLAG
  2 PRODUCT-ID
  2 QUANTITY
  2 TOTAL-PRICE
.
.
END-DEFINE
.
.
READ RECEIVABLE BY UNPAID-FLAG
.
.
  RESET INVOICE
  MOVE BY NAME RECEIVABLE TO INVOICE
.
  WRITE WORK FILE 1 INVOICE
.
END-READ
.
```

The output of a WRITE WORK FILE statement is a single record on a sequential file. It is considered good practice to define a record structure that can be referenced from the WRITE WORK FILE statement using the group name, as illustrated in the example above. Note however that individual fields can be explicitly written out, for example:

```
WRITE WORK FILE 1 /* INVOICE
  CLIENT.CLIENT-ID
  CLIENT.BILLING-NAME
```

The statements BACKOUT TRANSACTION and END TRANSACTION have no effect on sequential files. Once a WRITE WORK FILE statement has been executed, the record is irrevocably added to the output file. If restarted, a program may create duplicate records on an output sequential file.

On output sequential files, include fields which can be used to detect duplicates. Always include a sort step to eliminate duplicate records in any JCL that executes a NATURAL program with the potential to create such duplicates.

Exiting a routine or loop

ESCAPE statement

The ESCAPE statement allows you to exit a routine or processing loop, or to terminate an iteration of a processing loop. It has 3 forms, as follows:

- ESCAPE TOP
- ESCAPE BOTTOM
- ESCAPE ROUTINE

ESCAPE TOP and ESCAPE BOTTOM can be used in conjunction with any statement that creates a processing loop, such as FIND, FOR, HISTOGRAM, READ, READ WORK FILE, REPEAT, etc.

ESCAPE ROUTINE can be used in conjunction with a subroutine, subprogram or helproutine. It can also be used in conjunction with a program, where it acts as a STOP statement.

ESCAPE statement – ESCAPE TOP

ESCAPE TOP terminates the current iteration of a processing loop so that the next iteration can begin. Control passes to the top of the loop. Consider the following example:

```
DEFINE DATA
.
1 #INSTALLMENT-NBR (P2)
.
1 INVOICE
  2 CLIENT-ID (N8)
.
  2 NBR-INSTALLMENTS (N2)
  2 DUE-AMOUNT (N7.2/12)
.
.
END-DEFINE
.
.
READ WORK FILE 1 INVOICE
.
  FOR #INSTALLMENT-NBR = 1 TO INVOICE.NBR-INSTALLMENTS
    IF INVOICE.DUE-AMOUNT(#INSTALLMENT-NBR) = 0
      ESCAPE TOP
    END-IF
.
.
  END-FOR
.
END-WORK
.
```

In this example, ESCAPE TOP is used to exit a FOR loop and to simplify the logic of that loop. Without it, all of the additional FOR loop logic (not shown) would have to be conditioned by the IF statement.

ESCAPE statement – ESCAPE BOTTOM

ESCAPE BOTTOM terminates the current processing loop. Control passes to the statement following the bottom of the loop. Consider the following examples:

Example 1: ESCAPE BOTTOM

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
  2 UNPAID-FLAG
  2 PRODUCT-ID
.
END-DEFINE
.
READ RECEIVABLE BY CLIENT-ID
.
  IF RECEIVABLE.CLIENT-ID > 9999
    ESCAPE BOTTOM
  END-IF
.
.
.
END-READ
.
```

In this example, ESCAPE BOTTOM is used to exit a READ loop so that input records with a client ID over 9999 are not processed.

Example 2: ESCAPE BOTTOM with ESCAPE TOP

```
DEFINE DATA
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
  2 UNPAID-FLAG
  2 PRODUCT-ID
.
.
END-DEFINE
.
READ RECEIVABLE BY CLIENT-ID
.
  IF RECEIVABLE.CLIENT-ID > 9999
      ESCAPE BOTTOM
    END-IF
.
  IF RECEIVABLE.PRODUCT-ID NE 8108
      ESCAPE TOP
    END-IF
.
.
.
END-READ
.
```

In this example both ESCAPE TOP and ESCAPE BOTTOM are used within the same processing loop. ESCAPE BOTTOM is used to avoid processing records with a client ID over 9999. ESCAPE TOP is used to avoid processing records with a product id other than 8108.

When using both ESCAPE TOP and ESCAPE BOTTOM in the same processing loop, it's normally necessary to test for the conditions associated with ESCAPE BOTTOM before those associated with ESCAPE TOP, as per the above example.

Example 3: ESCAPE BOTTOM at label

```
DEFINE DATA
.
1 CLIENT VIEW OF CLIENT
  2 CLIENT-ID
.
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
.
END-DEFINE
.
.
  READ-RECEIVABLE.
READ RECEIVABLE BY CLIENT-ID
.
.
    FIND-CLIENT.
  FIND CLIENT WITH CLIENT-ID = RECEIVABLE.CLIENT-ID
    IF NO RECORD FOUND
      WRITE
        'CRITITCAL ERROR - CLIENT'
        RECEIVABLE.CLIENT-ID 'NOT FOUND'
      ESCAPE BOTTOM(READ-RECEIVABLE.)
    END-NOREC
.
  END-FIND
.
END-READ
.
.
```

In this example, the ESCAPE BOTTOM statement references a label for a processing loop other than the innermost in which it is coded. This would cause the referenced loop and all inner loops to terminate.

ESCAPE statement – ESCAPE ROUTINE

ESCAPE ROUTINE exits the current routine. The current routine may be a program, subprogram, subroutine or helproutine. In the case of a program, execution terminates completely. In the case of a subprogram, subroutine or helproutine, control passes to the statement following the “call”. In the case of a subroutine, control passes to the statement following the PERFORM.

Consider the following example:

```
DEFINE DATA
.
1 #CLIENT-ID (N8)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 CLIENT-ID
.
.
END-DEFINE
.
    PERFORM PROCESS-RECEIVABLE
.
.
DEFINE SUBROUTINE PROCESS-RECEIVABLE
************************************************************
** Processes one record per client from file RECEIVABLE.
************************************************************
  FIND-RECEIVABLE.
FIND (1) RECEIVABLE WITH CLIENT-ID = #CLIENT-ID
  IF NO RECORDS FOUND
    ESCAPE ROUTINE
  END-NOREC
.
.
END-FIND
.
END-SUBROUTINE /* PROCESS-RECEIVABLE
.
.
```

In this example, ESCAPE ROUTINE is used to exit a subroutine. The subroutine does nothing unless a record for the current client is found. When ESCAPE ROUTINE is executed, control passes to the statement following the PERFORM.

System variables and functions

This section describes essential system variables and system functions.

System variables provide access to system data. In general, they can be used in assignment statements, in conditional statements (e.g. IF, WHEN clause of DECIDE, UNTIL clause of REPEAT, etc.), in output statements (e.g. WRITE) and in formulas. System variables have names that start with an asterisk.

System functions manipulate system or user data. System functions have names that start with a letter.

The following are covered in this section of the course:

- *PF-KEY
- *DATX
- *TIMX
- *USER
- *PROGRAM
- *DEVICE
- VAL (value)
- SUBSTR (substring)

The following have already been covered in prior sections of the course:

- *PAGE-NUMBER
- *ISN
- *COUNTER
- *NUMBER
- POS (position)

System variable *PF-KEY

System variable *PF-KEY indicates how user input was terminated. Consider the following example:

```
INPUT
  USING MAP 'MAP1'
*
DECIDE ON FIRST VALUE OF *PF-KEY
  VALUE ‘PF3’
    ESCAPE ROUTINE
  VALUE ‘PF7’
    PERFORM SCROLL-UP
  VALUE ‘PF8’
    PERFORM SCROLL-DOWN
  VALUE ‘ENTR’
    PERFORM PROCESS-INPUT
  NONE
    REINPUT FULL
      WITH TEXT ‘Must terminate input using PF3/PF7/PF8 or ENTER’
      ALARM
END-DECIDE
.
```

In this example, an INPUT statement is immediately followed by logic to determine how input was terminated. Different actions are taken based on the value returned in *PF-KEY.

Note that the PF3 key is represented by the literal value “PF3”, PF7 by “PF7”, and so on. The ENTER key is represented by the literal value “ENTR”. The value returned in *PF-KEY is in A4 format.

The DECIDE FOR FIRST statement provides an elegant framework for logic to test the value returned in *PF-KEY.

System variable *DATX

System variable *DATX returns the current date in D (internal date) format. Consider the following example:

```
DEFINE DATA
.
LOCAL
.
1 #INDEX (P2)
.
1 RECEIVABLE VIEW OF RECEIVABLE
  2 C*BILL-DATE
  2 BILL-DATE (1:12)
.
END-DEFINE
.
.
FOR #INDEX = 1 TO RECEIVABLE.C*BILL-DATE
  DECIDE FOR FIRST CONDITION
    WHEN RECEIVABLE.BILL-DATE(#INDEX) > *DATX
*                    /* installment not yet due
      ESCAPE BOTTOM
.
  END-DECIDE
END-FOR
.
```

In this example RECEIVABLE.BILL-DATE(#INDEX), which contains a date in D format, is compared to the current date. If greater (i.e. later) than the current date, exits the FOR loop.

System variable *TIMX

System variable *TIMX returns the current date and time in T (internal time) format. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 UPDATE-VIEW VIEW OF CLIENT
  2 CLIENT-ID
  2 BILLING-NAME
  2 BILLING-ADDRESS (1:3)
  2 AUDIT-TIME
.
END-DEFINE
.
.
UPDATE-VIEW.AUDIT-TIME := *TIMX
.
```

In this example UPDATE-VIEW.AUDIT-TIME, which is a format T database field, is populated with the current date and time.

System variable *USER

System variable *USER returns the id of the current user in A8 format. For modules executing on-line, this is the current user's logon id. For modules executing in batch mode, this is the name of the batch job. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 UPDATE-VIEW VIEW OF CLIENT
  2 CLIENT-ID
  2 BILLING-NAME
  2 BILLING-ADDRESS (1:3)
  2 AUDIT-TIME
  2 AUDIT-USER
.
END-DEFINE
.
.
UPDATE-VIEW.AUDIT-USER := *USER
.
```

In this example UPDATE-VIEW.AUDIT-USER, which is a format A8 database field, is populated with the current user id.

System variable *PROGRAM

System variable *PROGRAM returns the name of the currently executing module in A8 format. Consider the following example:

```
DEFINE DATA
LOCAL
.
1 #PROGRAM (A8) INIT<*PROGRAM>
.
1 UPDATE-VIEW VIEW OF CLIENT
  2 CLIENT-ID
  2 BILLING-NAME
  2 BILLING-ADDRESS (1:3)
  2 AUDIT-TIME
  2 AUDIT-USER
  2 AUDIT-PROGRAM
.
END-DEFINE
.
.
UPDATE-VIEW.AUDIT-PROGRAM := #PROGRAM
.
```

In this example a work variable, #PROGRAM, is initialized with the name of the currently executing module. Subsequently, the stored value is used to populate UPDATE-VIEW.AUDIT-PROGRAM.

In modules where a system variable with an unchanging value needs to be referenced from multiple source lines, it's most efficient to obtain the value once and store it in a work variable.

System variable *DEVICE

System variable *DEVICE returns a value (in A8 format) indicating whether the current module is executing in batch mode. A value of “BATCH” indicates batch mode. Any other value indicates on-line mode. Consider the following example:

```
DEFINE DATA
.
END-DEFINE
.
.
IF *DEVICE = ‘BATCH’
  WRITE ‘Fatal error - check program logic’
  STOP
END-IF
.
```

System function VAL (value)

System function VAL (value) returns a numeric value derived from an alpha string containing only valid numeric digits. Consider the following example:

```
DEFINE DATA
.
LOCAL
.
1 #CODE (A1)
1 #COUNT (P5)
1 #INDEX (P3)
.
END-DEFINE
.
.
IF #CODE = MASK(N)
  #INDEX := 10 + VAL(#CODE)
  ADD 1 TO #COUNT(#INDEX)
END-IF
.
```

In this example, 10 is added to a numeric value extracted from variable #CODE and used as an index into an array.

System function SUBSTR (substring)

System function SUBSTR (substring) returns part of a string. Consider the following example:

```
DEFINE DATA
.
LOCAL
.
1 #MSG (A75)
.
END-DEFINE
.
.
IF SUBSTR(#MSG,1,5) = 'Error'
  REINPUT FULL WITH TEXT #MSG
    MARK *#MAP.#ACTION
    ALARM
.
END-IF
.
```

In this example, a substring of work variable #MSG is returned. Up to 5 characters starting from character position 1 are compared to the literal string ‘Error’.

Summary

The following tables summarize the system variables and system functions covered in this course:

System variables		
System Variable	**Format**	**Description**
*COUNTER	P10	Number of records presented in a READ, FIND or HISTOGRAM processing loop.
*DATX	D	Current date in internal date format.
*DEVICE	A8	A value indicating whether the current module is executing in batch mode.
*ISN	P10	Internal Sequence Number of a previously accessed database record.
*NUMBER	P10	Number of records with a specific key value.
*PAGE-NUMBER	P5	Current page number for a report.
*PF-KEY	A4	Coded value indicating how user input was terminated.
*PROGRAM	A8	Name of the currently executing module.
*TIMX	T	Current date and time in internal time format.
*USER	A8	ID of the current batch job or on-line user.

System functions	
System Function	**Description**
POS (position)	Returns the number (relative position) of a specified map field.
SUBSTR (substring)	Returns part of a string.
VAL (value)	Returns a numeric value derived from an alpha variable.

Syntax summary

This summary is designed to complement the necessarily complex, but rather cryptic, syntax diagrams published by Software AG. It covers the most commonly used NATURAL syntax plus, for reference purposes, Reporting Mode syntax and some statements and syntactical variations not covered in the preceding text. A definition of terms can be found at the end. A dynamic syntax look-up facility is also available on-line at www.spsimpson.com/natural/syntax.

Any underlined element (or group of elements) may be repeated. Some statements and syntactical variations have been deliberately omitted. Consult vendor documentation for complete language specifications.

Structured Mode

```
ACCEPT IF <logical condition>
ADD <type 20 operand> TO <type 48 operand>
ADD ROUNDED <type 20 operand> TO <type 48 operand>
ADD <type 20 operand> GIVING <type 47 operand>
ADD ROUNDED <type 20 operand> GIVING <type 47 operand>
ASSIGN <type 46 operand> = <type 21 operand>
ASSIGN ROUNDED <type 46 operand> = <type 21 operand>
ASSIGN <type 46 operand> = <arithmetic expression>
ASSIGN ROUNDED <type 46 operand> = <arithmetic expression>
<type 46 operand> := <type 21 operand>
<type 46 operand> := <arithmetic expression>
AT BREAK OF <type 28 operand>
  <statement>
END-BREAK
AT BREAK OF <type 30 operand> /<character positions>/
  <statement>
END-BREAK
AT BREAK (<label>) OF <type 28 operand>
  <statement>
END-BREAK
AT END OF DATA
  <statement>
END-ENDDATA
AT END OF DATA (<label>)
  <statement>
END-ENDDATA
AT END OF PAGE
  <statement>
END-ENDPAGE
AT END OF PAGE (<report number>)
  <statement>
END-ENDPAGE
```

```
AT START OF DATA
  <statement>
END-START
AT START OF DATA (<label>)
  <statement>
END-START
AT TOP OF PAGE
  <statement>
END-ENDPAGE
AT TOP OF PAGE (<report number>)
  <statement>
END-ENDPAGE
BACKOUT TRANSACTION
BEFORE BREAK PROCESSING
  <statement>
END-BEFORE
CALL <non-NATURAL subroutine>
CALL <non-NATURAL subroutine> <type 14 operand>
CALLNAT <subprogram name>
CALLNAT <subprogram name> <type 14 operand>
CALLNAT <subprogram name>
  <type 14 operand> <field modification attribute>
CLOSE WORK FILE <work file number>
COMPRESS <type 16 operand> INTO <type 27 operand>
COMPRESS <type 16 operand> INTO <type 27 operand> LEAVING NO SPACE
COMPRESS <type 16 operand> INTO <type 27 operand> DELIMITER <type 2 operand>
COMPUTE <type 46 operand> = <type 21 operand>
COMPUTE ROUNDED <type 46 operand> = <type 21 operand>
COMPUTE <type 46 operand> = <arithmetic expression>
COMPUTE ROUNDED <type 46 operand> = <arithmetic expression>
```

```
DECIDE FOR EVERY CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
DECIDE FOR EVERY CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN ANY
    <statement>
  WHEN ALL
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
DECIDE FOR FIRST CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
DECIDE FOR FIRST CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN ANY
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
```

```
DECIDE ON EVERY VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>
    <statement>
  NONE
    <statement>
END-DECIDE
DECIDE ON EVERY VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>, <type 12 operand>
    <statement>
  VALUE <type 12 operand>:<type 12 operand>, <type 12 operand>
    <statement>
  ANY
    <statement>
  ALL
    <statement>
  NONE
    <statement>
END-DECIDE
DECIDE ON FIRST VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>
    <statement>
  NONE
    <statement>
END-DECIDE
DECIDE ON FIRST VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>, <type 12 operand>
    <statement>
  VALUE <type 12 operand>:<type 12 operand>, <type 12 operand>
    <statement>
  ANY
    <statement>
  NONE
    <statement>
END-DECIDE
```

```
DEFINE DATA
GLOBAL USING <gda name>
LOCAL USING <lda name>
LOCAL USING <pda name>
LOCAL
  <in-line data definition>
END-DEFINE
```

```
DEFINE DATA
PARAMETER USING <pda name>
PARAMETER
  <in-line data definition>
LOCAL USING <lda name>
LOCAL USING <pda name>
LOCAL
  <in-line data definition>
END-DEFINE
```

```
DEFINE SUBROUTINE <subroutine name>
<statement>
END-SUBROUTINE
```

```
DEFINE WINDOW <window name>
```

```
DEFINE WINDOW <window name>
  SIZE <type 7 operand> * <type 7 operand>
  BASE <type 7 operand> / <type 7 operand>
```

```
DEFINE WINDOW <window name>
  SIZE <type 7 operand> * <type 7 operand>
  BASE <type 7 operand> / <type 7 operand>
  REVERSED
  TITLE <type 2 operand>
  CONTROL WINDOW
  FRAMED ON
```

```
DELETE
DELETE (<label>)
```

```
DISPLAY <type 44 operand>
DISPLAY (<report number>) <type 44 operand>
DISPLAY (<parameter assignment>) <type 44 operand>
DISPLAY NOTITLE <type 44 operand>
DISPLAY NOHDR <type 44 operand>
DISPLAY '=' <type 44 operand>
DISPLAY <text> <type 44 operand>
DISPLAY <text> '=' <type 44 operand>
DISPLAY <repeated character> <type 44 operand>
DISPLAY <repeated character> '=' <type 44 operand>
DISPLAY
  '/' <type 44 operand>
  '/' <type 44 operand>
DISPLAY
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> '=' <type 44 operand>
  <spacer> <repeated character> '=' <type 44 operand>
DISPLAY
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> '=' <type 44 operand>
  <tab> <repeated character> '=' <type 44 operand>
DIVIDE <type 18 operand> INTO <type 19 operand>
DIVIDE ROUNDED <type 18 operand> INTO <type 19 operand>
DIVIDE <type 18 operand> INTO <type 18 operand> GIVING <type 39 operand>
DIVIDE ROUNDED <type 18 operand> INTO <type 18 operand> GIVING <type 39 operand>
DIVIDE <type 18 operand> INTO <type 18 operand>
  GIVING <type 39 operand> REMAINDER <type 39 operand>
END
.
END TRANSACTION
END TRANSACTION <type 24 operand>
ESCAPE BOTTOM
ESCAPE BOTTOM IMMEDIATE
ESCAPE BOTTOM (<label>)
ESCAPE BOTTOM (<label>) IMMEDIATE
ESCAPE ROUTINE
ESCAPE ROUTINE IMMEDIATE
ESCAPE TOP
```

```
EXAMINE <type 9 operand> FOR <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand> DELIMITERS
  GIVING POSITION IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand> DELIMITER <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE FULL <type 9 operand> FOR <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE <type 9 operand> FOR FULL <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  GIVING LENGTH IN <type 32 operand>
EXAMINE FULL <type 51 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  GIVING INDEX IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand> AND DELETE
EXAMINE <type 9 operand> FOR <type 2 operand> AND DELETE FIRST
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand> AND DELETE FIRST
  GIVING LENGTH IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand>
  AND REPLACE WITH <type 2 operand>
EXAMINE <type 9 operand> FOR <type 2 operand>
  AND REPLACE FIRST WITH <type 2 operand>
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  AND REPLACE FIRST WITH FULL <type 2 operand>
  GIVING LENGTH IN <type 32 operand>
EXAMINE <type 35 operand> TRANSLATE INTO UPPER CASE
EXAMINE <type 35 operand> TRANSLATE INTO LOWER CASE
EXAMINE <type 35 operand> TRANSLATE USING <type 52 operand>
EXAMINE <type 35 operand> TRANSLATE USING INVERTED <type 52 operand>
FETCH <program name>
FETCH <program name> <type 14 operand>
FETCH RETURN <program name>
FETCH RETURN <program name> <type 14 operand>
```

```
FIND <view name> WITH <search criterion>
END-FIND
FIND (<limit>) <view name> WITH <search criterion>
END-FIND
FIND <view name> WITH <search criterion>
  RETAIN AS <set name>
END-FIND
FIND (<limit>) <view name> WITH <search criterion>
  RETAIN AS <set name>
END-FIND
FIND <view name> WITH <search criterion>
  SORTED BY <descriptor 1> <descriptor 2> <descriptor 3>
END-FIND
FIND <view name> WITH <search criterion>
  SORTED BY <descriptor 1> <descriptor 2> <descriptor 3> DESCENDING
END-FIND
FIND <view name> WITH <search criterion>
  IF NO RECORDS FOUND
    <statement>
  END-NOREC
END-FIND
FIND NUMBER <view name> WITH <search criterion>
FOR <type 31 operand> = <type 26 operand> THRU <type 26 operand>
  <statement>
END-FOR
FOR <type 31 operand> = <type 26 operand> THRU <type 26 operand>
    STEP <type 26 operand>
  <statement>
END-FOR
FORMAT <parameter assignment>
FORMAT (<report number>) <parameter assignment>
GET <view name> <type 25 operand>
GET <view name> *ISN (<label>)
GET SAME
GET SAME (<label>)
GET TRANSACTION DATA <type 29 operand>
HISTOGRAM <histogram view name> FOR <type 28 operand>
END-HISTOGRAM
HISTOGRAM (<limit>) <histogram view name> FOR <type 28 operand>
END-HISTOGRAM
HISTOGRAM <histogram view name> FOR <type 28 operand>
  FROM <type 4 operand>
END-HISTOGRAM
HISTOGRAM <histogram view name> FOR <type 28 operand>
  FROM <type 4 operand> THRU <type 4 operand>
END-HISTOGRAM
```

```
IF <logical condition>
  <statement>
END-IF
IF <logical condition>
  <statement>
ELSE
  <statement>
END-IF
IF <logical condition>
  IGNORE
ELSE
  <statement>
END-IF
IF SELECTION <type 40 operand>
  <statement>
ELSE
  <statement>
END-IF
IGNORE
INCLUDE <type 1 operand>
```

```
INPUT USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  USING MAP <map name>
INPUT
  WITH TEXT <message number>
  USING MAP <map name>
INPUT
  WITH TEXT <message number> ,<type 4 operand>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK <map field name>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <map field name>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK <type 13 operand>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <type 13 operand>
  USING MAP <map name>
INPUT
  WINDOW=<window name>
  WITH TEXT <type 3 operand>
  MARK <map field name>
  ALARM
  USING MAP <map name>
LIMIT <number of records>
MOVE <type 21 operand> TO <type 46 operand>
MOVE ROUNDED <type 21 operand> TO <type 46 operand>
MOVE SUBSTR(<type 9 operand>,<starting position>,<character positions>)
  TO <type 35 operand>
MOVE <type 9 operand>
  TO <SUBSTR(<type 35 operand>,<starting position>,<character positions>)
MOVE BY NAME <type 50 operand> TO <type 50 operand>
MOVE BY POSITION <type 50 operand> TO <type 50 operand>
MOVE EDITED <type 10 operand> TO <type 38 operand> (EM=<edit mask>)
MOVE EDITED <type 22 operand> (EM=<edit mask>) TO <type 36 operand>
MOVE LEFT <type 23 operand> TO <type 35 operand>
MOVE RIGHT <type 23 operand> TO <type 35 operand>
MOVE ALL <type 5 operand> TO <type 35 operand>
MOVE ALL <type 5 operand> TO <type 35 operand> UNTIL <type 6 operand>
```

```
MULTIPLY <type 49 operand> BY <type 18 operand>
MULTIPLY ROUNDED <type 49 operand> BY <type 18 operand>
MULTIPLY <type 18 operand> BY <type 18 operand> GIVING <type 49 operand>
MULTIPLY ROUNDED <type 18 operand> BY <type 18 operand> GIVING <type 49 operand>
NEWPAGE
NEWPAGE (<report number>)
NEWPAGE IF LESS THAN <type 7 operand> LINES LEFT
NEWPAGE EVEN IF TOP OF PAGE
ON ERROR
  <statement>
END-ERROR
PERFORM <subroutine name>
PERFORM <subroutine name> <type 14 operand>
PERFORM <subroutine name>
  <type 14 operand> <field modification attribute>
PERFORM BREAK PROCESSING
  <AT BREAK statement>
PRINT <type 44 operand>
PRINT (<report number>) <type 44 operand>
PRINT (<parameter assignment>) <type 44 operand>
PRINT NOTITLE <type 44 operand>
PRINT '=' <type 44 operand>
PRINT <text> <type 44 operand>
PRINT <repeated character> <type 44 operand>
PRINT
  '/' <type 44 operand>
  '/' <type 44 operand>
PRINT
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> <type 44 operand>
  <spacer> <repeated character> <type 44 operand>
PRINT
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> <type 44 operand>
  <tab> <repeated character> <type 44 operand>
```

```
READ <view name>
END-READ
READ (<limit>) <view name>
END-READ
READ <view name> IN PHYSICAL SEQUENCE
END-READ
READ <view name> BY ISN
END-READ
READ <view name> BY ISN
  FROM <type 4 operand>
END-READ
READ <view name> BY ISN
  THRU <type 4 operand>
END-READ
READ <view name> BY ISN
  FROM <type 4 operand> THRU <type 4 operand>
END-READ
READ <view name> BY <descriptor or superdescriptor>
END-READ
READ <view name> BY <descriptor or superdescriptor>
  FROM <type 4 operand>
END-READ
READ <view name> BY <descriptor>
  THRU <type 4 operand>
END-READ
READ <view name> BY <descriptor>
  FROM <type 4 operand> THRU <type 4 operand>
END-READ
READ WORK FILE <work file number> <type 42 operand>
  <statement>
END-WORK
READ WORK FILE <work file number> ONCE <type 42 operand>
  AT END OF FILE
    <statement>
  END-ENDFILE
READ WORK FILE <work file number> <type 42 operand>
    GIVING LENGTH IN <type 34 operand>
  <statement>
END-WORK
READ WORK FILE <work file number> ONCE <type 42 operand>
    GIVING LENGTH IN <type 34 operand>
  AT END OF FILE
    <statement>
  END-ENDFILE
```

```
REINPUT
  WITH TEXT <type 3 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
REINPUT FULL (<parameter assignment>)
  WITH TEXT <type 3 operand>
REINPUT FULL
  WITH TEXT <message number>
REINPUT FULL
  WITH TEXT <message number> ,<type 4 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK <map field name>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <map field name>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK <type 13 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <type 13 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK <map field name>
  ALARM
REINPUT FULL
  USING HELP
  MARK <map field name>
  ALARM
REJECT IF <logical condition>
RELEASE STACK
RELEASE SETS
RELEASE SETS <set name>
RELEASE VARIABLES
REPEAT
  <statement>
END-REPEAT
REPEAT
  <statement>
UNTIL <logical condition>
END-REPEAT
REPEAT
WHILE <logical condition>
  <statement>
END-REPEAT
```

```
RESET <type 42 operand>
RESET INITIAL <type 42 operand>
RETRY
SEPARATE <type 2 operand> INTO <type 41 operand>
SEPARATE <type 2 operand>
  LEFT JUSTIFIED INTO <type 41 operand>
SEPARATE <type 2 operand> INTO <type 41 operand>
  REMAINDER <type 27 operand>
SEPARATE <type 2 operand> INTO <type 41 operand> IGNORE
SEPARATE <type 2 operand> INTO <type 41 operand>
  WITH DELIMITER <type 2 operand>
SEPARATE <type 2 operand> INTO <type 41 operand>
  WITH RETAINED DELIMITER <type 2 operand>
SEPARATE <type 2 operand> INTO <type 41 operand>
  GIVING NUMBER IN <type 32 operand>
SEPARATE <type 2 operand> LEFT JUSTIFIED INTO <type 41 operand>
  REMAINDER <type 27 operand>
  WITH RETAINED DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
SEPARATE <type 2 operand> LEFT JUSTIFIED INTO <type 41 operand> IGNORE
  WITH RETAINED DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
SEPARATE SUBSTR(<type 2 operand>,<starting position>,<character positions>)
  LEFT JUSTIFIED INTO <type 41 operand> IGNORE
  WITH RETAINED DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
```

```
SET KEY ALL
SET KEY OFF
SET KEY COMMAND ON
SET KEY COMMAND OFF
SET KEY NAMED OFF
SET KEY <key>
SET KEY DYNAMIC <type 27 operand>
SET KEY <key> = OFF
SET KEY <key> = COMMAND
SET KEY DYNAMIC <type 27 operand> = COMMAND
SET KEY <key> = COMMAND OFF
SET KEY <key> NAMED <type 2 operand>
SET KEY DYNAMIC <type 27 operand> NAMED <type 2 operand>
SET KEY <key> NAMED OFF
SET KEY ENTR NAMED <type 2 operand>
SET KEY ENTR NAMED OFF
SET KEY <key> = HELP
SET KEY DYNAMIC <type 27 operand> = HELP
SET KEY <key> = HELP NAMED <type 2 operand>
SET KEY <key> = HELP NAMED OFF
SET KEY <key> = <type 2 operand>
SET KEY DYNAMIC <type 27 operand> = <type 2 operand>
SET KEY <key> = <type 2 operand> NAMED <type 2 operand>
SET KEY <key> = <type 2 operand> NAMED OFF
SET KEY <key> = DATA <type 2 operand>
SET KEY DYNAMIC <type 27 operand> = DATA <type 2 operand>
SET KEY <key> = DATA <type 2 operand> NAMED <type 2 operand>
SET KEY <key> = DATA <type 2 operand> NAMED OFF
<label>
SET TIME
<label>
SETTIME
SET WINDOW <window name>
SET WINDOW OFF
SKIP <type 7 operand> LINES
SKIP (<report number>) <type 7 operand> LINES
```

```
END-ALL
SORT BY <type 29 operand> USING <type 37 operand>
  <statement>
END-SORT
END-ALL
SORT BY <type 29 operand> USING KEYS
  <statement>
END-SORT
END-ALL
SORT BY <type 29 operand> DESCENDING USING <type 37 operand>
  <statement>
END-SORT
END-ALL
SORT BY <type 29 operand> DESCENDING USING KEYS
  <statement>
END-SORT
END-ALL
SORT BY
    <type 29 operand> ASCENDING
    <type 29 operand> DESCENDING
    <type 29 operand> ASCENDING
    USING <type 37 operand>
  <statement>
END-SORT
END-ALL
SORT BY
    <type 29 operand> ASCENDING
    <type 29 operand> DESCENDING
    <type 29 operand> ASCENDING
    USING KEYS
  <statement>
END-SORT
STACK <type 17 operand>
STACK FORMATTED <type 17 operand>
STACK COMMAND <type 15 operand>
STACK COMMAND <type 15 operand> <type 17 operand>
STACK TOP <type 17 operand>
STACK TOP FORMATTED <type 17 operand>
STACK TOP COMMAND <type 15 operand>
STACK TOP COMMAND <type 15 operand> <type 17 operand>
STOP
STORE <view name>
STORE <view name> USING NUMBER <type 33 operand>
```

```
SUBTRACT <type 20 operand> FROM <type 48 operand>
SUBTRACT ROUNDED <type 20 operand> FROM <type 48 operand>
SUBTRACT <type 20 operand> FROM <type 48 operand>
  GIVING <type 47 operand>
SUBTRACT ROUNDED <type 20 operand> FROM <type 48 operand>
  GIVING <type 47 operand>
SUSPEND IDENTICAL SUPPRESS
SUSPEND IDENTICAL SUPPRESS <report number>
TERMINATE
TERMINATE <type 8 operand>
TERMINATE <type 8 operand> <type 11 operand>
UPDATE
UPDATE (<label>)
WRITE <type 44 operand>
WRITE (<report number>) <type 44 operand>
WRITE (<parameter assignment>) <type 44 operand>
WRITE NOTITLE <type 44 operand>
WRITE '=' <type 44 operand>
WRITE <text> <type 44 operand>
WRITE <text> '=' <type 44 operand>
WRITE <repeated character> <type 44 operand>
WRITE <repeated character> '=' <type 44 operand>
WRITE
  '/' <type 44 operand>
  '/' <type 44 operand>
WRITE
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> '=' <type 44 operand>
  <spacer> <repeated character> '=' <type 44 operand>
WRITE
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> '=' <type 44 operand>
  <tab> <repeated character> '=' <type 44 operand>
WRITE USING FORM <form name>
WRITE USING FORM <form name> <type 44 operand>
WRITE USING MAP <form name>
WRITE USING MAP <form name> <type 44 operand>
```

```
WRITE TITLE <type 44 operand>
WRITE (<report number>) TITLE <type 44 operand>
WRITE TITLE LEFT JUSTIFIED <type 44 operand>
WRITE TITLE UNDERLINED <type 44 operand>
WRITE TITLE (<parameter assignment>) <type 44 operand>
WRITE TITLE '=' <type 44 operand>
WRITE TITLE <text> <type 44 operand>
WRITE TITLE <repeated character> <type 44 operand>
WRITE TITLE
  '/' <type 44 operand>
  '/' <type 44 operand>
WRITE TITLE
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> <type 44 operand>
  <spacer> <repeated character> <type 44 operand>
WRITE TITLE
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> <type 44 operand>
  <tab> <repeated character> <type 44 operand>
WRITE TITLE
  <tab> <type 44 operand>
  SKIP <type 6 operand> LINES
```

```
WRITE TRAILER <type 44 operand>
WRITE (<report number>) TRAILER <type 44 operand>
WRITE TRAILER LEFT JUSTIFIED <type 44 operand>
WRITE TRAILER UNDERLINED <type 44 operand>
WRITE TRAILER (<parameter assignment>) <type 44 operand>
WRITE TRAILER '=' <type 44 operand>
WRITE TRAILER <text> <type 44 operand>
WRITE TRAILER <repeated character> <type 44 operand>
WRITE TRAILER
  '/' <type 44 operand>
  '/' <type 44 operand>
WRITE TRAILER
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> <type 44 operand>
  <spacer> <repeated character> <type 44 operand>
WRITE TRAILER
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> <type 44 operand>
  <tab> <repeated character> <type 44 operand>
WRITE TRAILER
  <tab> <type 44 operand>
  SKIP <type 6 operand> LINES
WRITE WORK FILE <work file number> <type 14 operand>
```

Reporting Mode

```
ACCEPT IF <logical condition>
ADD <type 20 operand> TO <type 48 operand>
ADD ROUNDED <type 20 operand> TO <type 48 operand>
ADD <type 20 operand> GIVING <type 47 operand>
ADD ROUNDED <type 20 operand> GIVING <type 47 operand>
ASSIGN <type 46 operand> = <type 21 operand>
ASSIGN ROUNDED <type 46 operand> = <type 21 operand>
ASSIGN <type 46 operand> = <arithmetic expression>
ASSIGN ROUNDED <type 46 operand> = <arithmetic expression>
<type 46 operand> = <type 21 operand>
<type 46 operand> = <arithmetic expression>
AT BREAK OF <type 28 operand>
DO
  <statement>
DOEND
AT BREAK OF <type 30 operand> /<character positions>/
DO
  <statement>
DOEND
AT BREAK (<label>) OF <type 28 operand>
DO
  <statement>
DOEND
AT END OF DATA
DO
  <statement>
DOEND
AT END OF DATA (<label>)
DO
  <statement>
DOEND
AT END OF PAGE
DO
  <statement>
DOEND
AT END OF PAGE (<report number>)
DO
  <statement>
DOEND
```

```
AT START OF DATA
DO
  <statement>
DOEND
AT START OF DATA (<label>)
DO
  <statement>
DOEND
AT TOP OF PAGE
DO
  <statement>
DOEND
AT TOP OF PAGE (<report number>)
DO
  <statement>
DOEND
BACKOUT TRANSACTION
BEFORE BREAK PROCESSING
DO
  <statement>
DOEND
CALL <non-NATURAL subroutine>
CALL <non-NATURAL subroutine> <type 14 operand>
CALLNAT <subprogram name>
CALLNAT <subprogram name> <type 14 operand>
CALLNAT <subprogram name>
  <type 14 operand> <field modification attribute>
CLOSE WORK FILE <work file number>
COMPRESS <type 16 operand> INTO <type 27 operand>
COMPRESS <type 16 operand> INTO <type 27 operand> LEAVING NO SPACE
COMPRESS <type 16 operand> INTO <type 27 operand> DELIMITER <type 2 operand>
COMPUTE <type 46 operand> = <type 21 operand>
COMPUTE ROUNDED <type 46 operand> = <type 21 operand>
COMPUTE <type 46 operand> = <arithmetic expression>
COMPUTE ROUNDED <type 46 operand> = <arithmetic expression>
```

```
DECIDE FOR EVERY CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
DECIDE FOR EVERY CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN ANY
    <statement>
  WHEN ALL
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
DECIDE FOR FIRST CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
DECIDE FOR FIRST CONDITION
  WHEN <logical condition>
    <statement>
  WHEN <logical condition>
    <statement>
  WHEN ANY
    <statement>
  WHEN NONE
    <statement>
END-DECIDE
```

```
DECIDE ON EVERY VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>
    <statement>
  NONE
    <statement>
END-DECIDE
DECIDE ON EVERY VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>, <type 12 operand>
    <statement>
  VALUE <type 12 operand>:<type 12 operand>, <type 12 operand>
    <statement>
  ANY
    <statement>
  ALL
    <statement>
  NONE
    <statement>
END-DECIDE
DECIDE ON FIRST VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>
    <statement>
  NONE
    <statement>
END-DECIDE
DECIDE ON FIRST VALUE OF <type 45 operand>
  VALUE <type 12 operand>
    <statement>
  VALUE <type 12 operand>, <type 12 operand>
    <statement>
  VALUE <type 12 operand>:<type 12 operand>, <type 12 operand>
    <statement>
  ANY
    <statement>
  NONE
    <statement>
END-DECIDE
```

```
DEFINE DATA
GLOBAL USING <gda name>
LOCAL USING <lda name>
LOCAL USING <pda name>
LOCAL
  <in-line data definition>
END-DEFINE
DEFINE DATA
PARAMETER USING <pda name>
PARAMETER
  <in-line data definition>
LOCAL USING <lda name>
LOCAL USING <pda name>
LOCAL
  <in-line data definition>
END-DEFINE
DEFINE SUBROUTINE <subroutine name>
<statement>
END-SUBROUTINE
DEFINE SUBROUTINE <subroutine name>
<statement>
RETURN
DEFINE WINDOW <window name>
DEFINE WINDOW <window name>
  SIZE <type 7 operand> * <type 7 operand>
  BASE <type 7 operand> / <type 7 operand>
DEFINE WINDOW <window name>
  SIZE <type 7 operand> * <type 7 operand>
  BASE <type 7 operand> / <type 7 operand>
  REVERSED
  TITLE <type 2 operand>
  CONTROL WINDOW
  FRAMED ON
DELETE
DELETE (<label>)
```

```
DISPLAY <type 44 operand>
DISPLAY (<report number>) <type 44 operand>
DISPLAY (<parameter assignment>) <type 44 operand>
DISPLAY NOTITLE <type 44 operand>
DISPLAY NOHDR <type 44 operand>
DISPLAY '=' <type 44 operand>
DISPLAY <text> <type 44 operand>
DISPLAY <text> '=' <type 44 operand>
DISPLAY <repeated character> <type 44 operand>
DISPLAY <repeated character> '=' <type 44 operand>
DISPLAY
  '/' <type 44 operand>
  '/' <type 44 operand>
DISPLAY
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> '=' <type 44 operand>
  <spacer> <repeated character> '=' <type 44 operand>
DISPLAY
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> '=' <type 44 operand>
  <tab> <repeated character> '=' <type 44 operand>
DIVIDE <type 18 operand> INTO <type 19 operand>
DIVIDE ROUNDED <type 18 operand> INTO <type 19 operand>
DIVIDE <type 18 operand> INTO <type 18 operand> GIVING <type 39 operand>
DIVIDE ROUNDED <type 18 operand> INTO <type 18 operand> GIVING <type 39 operand>
DIVIDE <type 18 operand> INTO <type 18 operand>
  GIVING <type 39 operand> REMAINDER <type 39 operand>
END
.
END TRANSACTION
END TRANSACTION <type 24 operand>
ESCAPE
ESCAPE BOTTOM
ESCAPE BOTTOM IMMEDIATE
ESCAPE BOTTOM (<label>)
ESCAPE BOTTOM (<label>) IMMEDIATE
ESCAPE ROUTINE
ESCAPE ROUTINE IMMEDIATE
ESCAPE TOP
```

```
EXAMINE <type 9 operand> FOR <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand> DELIMITERS
  GIVING POSITION IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand> DELIMITER <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE FULL <type 9 operand> FOR <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE <type 9 operand> FOR FULL <type 2 operand>
  GIVING POSITION IN <type 32 operand>
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  GIVING LENGTH IN <type 32 operand>
EXAMINE FULL <type 51 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  GIVING INDEX IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand> AND DELETE
EXAMINE <type 9 operand> FOR <type 2 operand> AND DELETE FIRST
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand> AND DELETE FIRST
  GIVING LENGTH IN <type 32 operand>
EXAMINE <type 9 operand> FOR <type 2 operand>
  AND REPLACE WITH <type 2 operand>
EXAMINE <type 9 operand> FOR <type 2 operand>
  AND REPLACE FIRST WITH <type 2 operand>
EXAMINE FULL <type 9 operand> FOR FULL <type 2 operand>
  DELIMITER <type 2 operand>
  AND REPLACE FIRST WITH FULL <type 2 operand>
  GIVING LENGTH IN <type 32 operand>
EXAMINE <type 35 operand> TRANSLATE INTO UPPER CASE
EXAMINE <type 35 operand> TRANSLATE INTO LOWER CASE
EXAMINE <type 35 operand> TRANSLATE USING <type 52 operand>
EXAMINE <type 35 operand> TRANSLATE USING INVERTED <type 52 operand>
FETCH <program name>
FETCH <program name> <type 14 operand>
FETCH RETURN <program name>
FETCH RETURN <program name> <type 14 operand>
```

```
FIND <view name> WITH <search criterion>
LOOP
FIND (<limit>) <view name> WITH <search criterion>
LOOP
FIND <view name> WITH <search criterion>
  RETAIN AS <set name>
LOOP
FIND <view name> WITH <search criterion>
  SORTED BY <descriptor 1> <descriptor 2> <descriptor 3>
LOOP
FIND <view name> WITH <search criterion>
  SORTED BY <descriptor 1> <descriptor 2> <descriptor 3> DESCENDING
LOOP
FIND <view name> WITH <search criterion>
  IF NO RECORDS FOUND
  DO
    <statement>
  DOEND
LOOP
FIND FIRST <view name> WITH <search criterion>
FIND NUMBER <view name> WITH <search criterion>
FIND UNIQUE <view name> WITH <search criterion>
FOR <type 31 operand> = <type 26 operand> THRU <type 26 operand>
  <statement>
LOOP
FOR <type 31 operand> = <type 26 operand> THRU <type 26 operand>
    STEP <type 26 operand>
  <statement>
LOOP
FORMAT <parameter assignment>
FORMAT (<report number>) <parameter assignment>
GET <view name> <type 25 operand>
GET <view name> *ISN (<label>)
GET SAME
GET SAME (<label>)
GET SAME (<label>) <type 53 operand>
GET SAME <type 53 operand>
GET TRANSACTION DATA <type 29 operand>
```

```
HISTOGRAM <histogram view name> FOR <type 28 operand>
LOOP
HISTOGRAM (<limit>) <histogram view name> FOR <type 28 operand>
LOOP
HISTOGRAM <histogram view name> FOR <type 28 operand>
  FROM <type 4 operand>
LOOP
HISTOGRAM <histogram view name> FOR <type 28 operand>
  FROM <type 4 operand> THRU <type 4 operand>
LOOP
IF <logical condition>
<statement>
IF <logical condition>
DO
  <statement>
DOEND
IF <logical condition>
<statement>
ELSE
<statement>
IF <logical condition>
DO
  <statement>
DOEND
ELSE
DO
  <statement>
DOEND
IF <logical condition>
DO
  <statement>
DOEND
ELSE
  <statement>
IF <logical condition>
IGNORE
ELSE
DO
  <statement>
DOEND
```

```
IF SELECTION <type 40 operand>
DO
  <statement>
DOEND
ELSE
DO
  <statement>
DOEND
IGNORE
INCLUDE <type 1 operand>
INPUT USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  USING MAP <map name>
INPUT
  WITH TEXT <message number>
  USING MAP <map name>
INPUT
  WITH TEXT <message number> ,<type 4 operand>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK <map field name>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <map field name>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK <type 13 operand>
  USING MAP <map name>
INPUT
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <type 13 operand>
  USING MAP <map name>
INPUT
  WINDOW=<window name>
  WITH TEXT <type 3 operand>
  MARK <map field name>
  ALARM
  USING MAP <map name>
LIMIT <number of records>
LOOP
CLOSE LOOP
```

```
MOVE <type 21 operand> TO <type 46 operand>
MOVE ROUNDED <type 21 operand> TO <type 46 operand>
MOVE SUBSTR(<type 9 operand>,<starting position>,<character positions>)
  TO <type 35 operand>
MOVE <type 9 operand>
  TO <SUBSTR(<type 35 operand>,<starting position>,<character positions>)
MOVE BY NAME <type 50 operand> TO <type 50 operand>
MOVE BY POSITION <type 50 operand> TO <type 50 operand>
MOVE EDITED <type 10 operand> TO <type 38 operand> (EM=<edit mask>)
MOVE EDITED <type 22 operand> (EM=<edit mask>) TO <type 36 operand>
MOVE LEFT <type 23 operand> TO <type 35 operand>
MOVE RIGHT <type 23 operand> TO <type 35 operand>
MOVE ALL <type 5 operand> TO <type 35 operand>
MOVE ALL <type 5 operand> TO <type 35 operand> UNTIL <type 6 operand>
MULTIPLY <type 49 operand> BY <type 18 operand>
MULTIPLY ROUNDED <type 49 operand> BY <type 18 operand>
MULTIPLY <type 18 operand> BY <type 18 operand> GIVING <type 49 operand>
MULTIPLY ROUNDED <type 18 operand> BY <type 18 operand> GIVING <type 49 operand>
NEWPAGE
NEWPAGE (<report number>)
NEWPAGE IF LESS THAN <type 7 operand> LINES LEFT
NEWPAGE EVEN IF TOP OF PAGE
OBTAIN <type 43 operand>
ON ERROR
DO
  <statement>
DOEND
PERFORM <subroutine name>
PERFORM <subroutine name> <type 14 operand>
PERFORM <subroutine name>
  <type 14 operand> <field modification attribute>
PERFORM BREAK PROCESSING
  <AT BREAK statement>
```

```
PRINT <type 44 operand>
PRINT (<report number>) <type 44 operand>
PRINT (<parameter assignment>) <type 44 operand>
PRINT NOTITLE <type 44 operand>
PRINT '=' <type 44 operand>
PRINT <text> <type 44 operand>
PRINT <repeated character> <type 44 operand>
PRINT
  '/' <type 44 operand>
  '/' <type 44 operand>
PRINT
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> <type 44 operand>
  <spacer> <repeated character> <type 44 operand>
PRINT
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> <type 44 operand>
  <tab> <repeated character> <type 44 operand>
```

```
READ <view name>
LOOP
READ (<limit>) <view name>
LOOP
READ <view name> IN PHYSICAL SEQUENCE
LOOP
READ <view name> BY ISN
LOOP
READ <view name> BY ISN
  FROM <type 4 operand>
LOOP
READ <view name> BY ISN
  THRU <type 4 operand>
LOOP
READ <view name> BY ISN
  FROM <type 4 operand> THRU <type 4 operand>
LOOP
READ <view name> BY <descriptor or superdescriptor>
LOOP
READ <view name> BY <descriptor or superdescriptor>
  FROM <type 4 operand>
LOOP
READ <view name> BY <descriptor>
  THRU <type 4 operand>
LOOP
READ <view name> BY <descriptor>
  FROM <type 4 operand> THRU <type 4 operand>
LOOP
READ WORK FILE <work file number> <type 42 operand>
  <statement>
LOOP
READ WORK FILE <work file number> ONCE <type 42 operand>
  AT END OF FILE
  DO
    <statement>
  DOEND
READ WORK FILE <work file number> <type 42 operand>
    GIVING LENGTH IN <type 34 operand>
  <statement>
LOOP
READ WORK FILE <work file number> ONCE <type 42 operand>
    GIVING LENGTH IN <type 34 operand>
  AT END OF FILE
  DO
    <statement>
  DOEND
```

```
REDEFINE <type 42 operand> (<type 42 operand> )
REDEFINE
  <type 42 operand> (<type 42 operand> )
  <type 42 operand> (<type 42 operand> )
REINPUT
  WITH TEXT <type 3 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
REINPUT FULL (<parameter assignment>)
  WITH TEXT <type 3 operand>
REINPUT FULL
  WITH TEXT <message number>
REINPUT FULL
  WITH TEXT <message number> ,<type 4 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK <map field name>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <map field name>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK <type 13 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK POSITION <type 7 operand> IN <type 13 operand>
REINPUT FULL
  WITH TEXT <type 3 operand>
  MARK <map field name>
  ALARM
REINPUT FULL
  USING HELP
  MARK <map field name>
  ALARM
REJECT IF <logical condition>
RELEASE STACK
RELEASE SETS
RELEASE SETS <set name>
RELEASE VARIABLES
```

```
REPEAT
  <statement>
LOOP
REPEAT
  <statement>
UNTIL <logical condition>
LOOP
REPEAT
WHILE <logical condition>
  <statement>
LOOP
RESET <type 42 operand> <format/length specification>
RESET <type 42 operand>
RESET INITIAL <type 42 operand>
RETRY
SEPARATE <type 2 operand> INTO <type 41 operand>
SEPARATE <type 2 operand>
  LEFT JUSTIFIED INTO <type 41 operand>
SEPARATE <type 2 operand> INTO <type 41 operand>
  REMAINDER <type 27 operand>
SEPARATE <type 2 operand> INTO <type 41 operand> IGNORE
SEPARATE <type 2 operand> INTO <type 41 operand>
  WITH DELIMITER <type 2 operand>
SEPARATE <type 2 operand> INTO <type 41 operand>
  WITH RETAINED DELIMITER <type 2 operand>
SEPARATE <type 2 operand> INTO <type 41 operand>
  GIVING NUMBER IN <type 32 operand>
SEPARATE <type 2 operand> LEFT JUSTIFIED INTO <type 41 operand>
  REMAINDER <type 27 operand>
  WITH RETAINED DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
SEPARATE <type 2 operand> LEFT JUSTIFIED INTO <type 41 operand> IGNORE
  WITH RETAINED DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
SEPARATE SUBSTR(<type 2 operand>,<starting position>,<character positions>)
  LEFT JUSTIFIED INTO <type 41 operand> IGNORE
  WITH RETAINED DELIMITER <type 2 operand>
  GIVING NUMBER IN <type 32 operand>
SET GLOBALS <parameter assignment>
```

```
SET KEY ALL
SET KEY OFF
SET KEY COMMAND ON
SET KEY COMMAND OFF
SET KEY NAMED OFF
SET KEY <key>
SET KEY DYNAMIC <type 27 operand>
SET KEY <key> = OFF
SET KEY <key> = COMMAND
SET KEY DYNAMIC <type 27 operand> = COMMAND
SET KEY <key> = COMMAND OFF
SET KEY <key> NAMED <type 2 operand>
SET KEY DYNAMIC <type 27 operand> NAMED <type 2 operand>
SET KEY <key> NAMED OFF
SET KEY ENTR NAMED <type 2 operand>
SET KEY ENTR NAMED OFF
SET KEY <key> = HELP
SET KEY DYNAMIC <type 27 operand> = HELP
SET KEY <key> = HELP NAMED <type 2 operand>
SET KEY <key> = HELP NAMED OFF
SET KEY <key> = <type 2 operand>
SET KEY DYNAMIC <type 27 operand> = <type 2 operand>
SET KEY <key> = <type 2 operand> NAMED <type 2 operand>
SET KEY <key> = <type 2 operand> NAMED OFF
SET KEY <key> = DATA <type 2 operand>
SET KEY DYNAMIC <type 27 operand> = DATA <type 2 operand>
SET KEY <key> = DATA <type 2 operand> NAMED <type 2 operand>
SET KEY <key> = DATA <type 2 operand> NAMED OFF
<label>
SET TIME
<label>
SETTIME
SET WINDOW <window name>
SET WINDOW OFF
SKIP <type 7 operand> LINES
SKIP (<report number>) <type 7 operand> LINES
```

```
SORT BY <type 29 operand> USING <type 37 operand>
  <statement>
SORT BY <type 29 operand> USING KEYS
  <statement>
SORT BY <type 29 operand> DESCENDING USING <type 37 operand>
  <statement>
SORT BY <type 29 operand> DESCENDING USING KEYS
  <statement>
SORT BY
    <type 29 operand> ASCENDING
    <type 29 operand> DESCENDING
    <type 29 operand> ASCENDING
    USING <type 37 operand>
  <statement>
SORT BY
    <type 29 operand> ASCENDING
    <type 29 operand> DESCENDING
    <type 29 operand> ASCENDING
    USING KEYS
  <statement>
STACK <type 17 operand>
STACK FORMATTED <type 17 operand>
STACK COMMAND <type 15 operand>
STACK COMMAND <type 15 operand> <type 17 operand>
STACK TOP <type 17 operand>
STACK TOP FORMATTED <type 17 operand>
STACK TOP COMMAND <type 15 operand>
STACK TOP COMMAND <type 15 operand> <type 17 operand>
STOP
STORE <view name>
STORE <view name>
  USING SAME RECORD (<label>)
STORE <view name> USING NUMBER <type 33 operand>
STORE <view name> USING NUMBER <type 33 operand>
  USING SAME RECORD (<label>)
STORE <view name> USING NUMBER <type 33 operand>
  WITH
    <in-line data assignment>
SUBTRACT <type 20 operand> FROM <type 48 operand>
SUBTRACT ROUNDED <type 20 operand> FROM <type 48 operand>
SUBTRACT <type 20 operand> FROM <type 48 operand>
  GIVING <type 47 operand>
SUBTRACT ROUNDED <type 20 operand> FROM <type 48 operand>
  GIVING <type 47 operand>
SUSPEND IDENTICAL SUPPRESS
SUSPEND IDENTICAL SUPPRESS <report number>
```

```
TERMINATE
TERMINATE <type 8 operand>
TERMINATE <type 8 operand> <type 11 operand>
UPDATE
UPDATE (<label>)
UPDATE (<label>)
  USING SAME RECORD
UPDATE (<label>)
  WITH
    <in-line data assignment>
WRITE <type 44 operand>
WRITE (<report number>) <type 44 operand>
WRITE (<parameter assignment>) <type 44 operand>
WRITE NOTITLE <type 44 operand>
WRITE '=' <type 44 operand>
WRITE <text> <type 44 operand>
WRITE <text> '=' <type 44 operand>
WRITE <repeated character> <type 44 operand>
WRITE <repeated character> '=' <type 44 operand>
WRITE
  '/' <type 44 operand>
  '/' <type 44 operand>
WRITE
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> '=' <type 44 operand>
  <spacer> <repeated character> '=' <type 44 operand>
WRITE
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> '=' <type 44 operand>
  <tab> <repeated character> '=' <type 44 operand>
WRITE USING FORM <form name>
WRITE USING FORM <form name> <type 44 operand>
WRITE USING MAP <form name>
WRITE USING MAP <form name> <type 44 operand>
```

```
WRITE TITLE <type 44 operand>
WRITE (<report number>) TITLE <type 44 operand>
WRITE TITLE LEFT JUSTIFIED <type 44 operand>
WRITE TITLE UNDERLINED <type 44 operand>
WRITE TITLE (<parameter assignment>) <type 44 operand>
WRITE TITLE '=' <type 44 operand>
WRITE TITLE <text> <type 44 operand>
WRITE TITLE <repeated character> <type 44 operand>
WRITE TITLE
  '/' <type 44 operand>
  '/' <type 44 operand>
WRITE TITLE
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> <type 44 operand>
  <spacer> <repeated character> <type 44 operand>
WRITE TITLE
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> <type 44 operand>
  <tab> <repeated character> <type 44 operand>
WRITE TITLE
  <tab> <type 44 operand>
  SKIP <type 6 operand> LINES
WRITE TRAILER <type 44 operand>
WRITE (<report number>) TRAILER <type 44 operand>
```

```
WRITE TRAILER LEFT JUSTIFIED <type 44 operand>
WRITE TRAILER UNDERLINED <type 44 operand>
WRITE TRAILER (<parameter assignment>) <type 44 operand>
WRITE TRAILER '=' <type 44 operand>
WRITE TRAILER <text> <type 44 operand>
WRITE TRAILER <repeated character> <type 44 operand>
WRITE TRAILER
  '/' <type 44 operand>
  '/' <type 44 operand>
WRITE TRAILER
  <spacer> <type 44 operand>
  <spacer> '=' <type 44 operand>
  <spacer> <text> <type 44 operand>
  <spacer> <repeated character> <type 44 operand>
WRITE TRAILER
  <tab> <type 44 operand>
  <tab> '=' <type 44 operand>
  <tab> <text> <type 44 operand>
  <tab> <repeated character> <type 44 operand>
WRITE TRAILER
  <tab> <type 44 operand>
  SKIP <type 6 operand> LINES
WRITE WORK FILE <work file number> <type 14 operand>
```

Definition of terms

<arithmetic expression>	A formula which is evaluated using standard mathematical rules, containing references to variables and constants joined by any of the following arithmetic operators: + (add), – (subtract), / (divide), * (multiply) or ** (exponent), optionally containing paired parentheses to control the evaluation.
<character positions>	Number of character positions.
<color definition attribute>	A string in the format CD=***X***, where ***X*** indicates the required color definition attribute. Valid values for ***X*** are BL (blue), GR (green), NE (neutral/white), PI (pink/magenta), RE (red), TU (turquoise/cyan), YE (yellow).
<descriptor>	Name of a descriptor field to be used for sequencing.
<descriptor or superdescriptor>	Name of a descriptor field or superdescriptor field to be used for sequencing.
<descriptor 1>	Descriptor field to be used as primary sort key.
<descriptor 2>	Descriptor field to be used as secondary sort key (optional).
<descriptor 3>	Descriptor field to be used as tertiary sort key (optional).
<empty field filler character>	A string in the format AD=***'x'***, where ***x*** represents the required empty field filler character (an underscore character by convention). The empty field filler character is a type of display attribute.
<field alignment attribute>	A string in the format AD=***X***, where ***X*** indicates the required field alignment attribute. Valid values for ***X*** are L (left-justified), R (right-justified), Z (right-justified with leading zeroes). L is the default for alpha fields, R is the default for numeric fields. Field alignment attributes are a type of display attribute.
<field case attribute>	A string in the format AD=***X***, where ***X*** indicates the required field case attribute. Valid values for ***X*** are T (translate to upper case), W (preserve lower case). Field case attributes are a type of display attribute.
<field input/output attribute>	A string in the format AD=***X***, where ***X*** indicates the required field input/output attribute. Valid values for ***X*** are A (input field, non-protected), M (output field, modifiable), O (output field, protected), P (temporarily protected). Field input/output attributes are a type of display attribute.
<field length attribute>	A string in the format AD=***X***, where ***X*** indicates the required field length attribute. Valid values for ***X*** are G (input data must fill the field) and H (input data need not fill the field). Field length attributes apply solely to input-only fields. Field length attributes are a type of display attribute.
<field mandatory attribute>	A string in the format AD=***X***, where ***X*** indicates the required field mandatory attribute. Valid values for ***X*** are E (null value is invalid), F (null value is valid). Field mandatory attributes apply solely to input-only fields. Field mandatory attributes are a type of display attribute.
<field modification attribute>	A string in the format AD=***X***, where ***X*** indicates the required field modification attribute. Valid values for ***X*** are M (output field, modifiable), O (output field, protected). Field modification attributes are a type of display attribute.

<field representation attribute>	A string in the format AD=***X***, where ***X*** indicates the required field representation attribute. Valid values for ***X*** are B (blinking), C (cursive/italic), D (default intensity), I (intensified), N (non-display), U (underlined), V (reverse video). Field representation attributes are a type of display attribute.
<form name>	Name of a NATURAL map module to be used in conjunction with a WRITE USING FORM statement.
<format/length specification>	Format specification or format and length specification enclosed in parentheses. Examples: (L)**,** (A8)**,** (P3.2)**.**
<histogram view name>	Name of a database view defining only the descriptor (or superdescriptor) referenced in the associated HISTOGRAM statement.
<in-line data assignment>	One or more in-line data assignment statements.
<in-line data definition>	One or more in-line data definition statements.
<label>	Statement label.
<key>	A key that, if active, can be used to terminate input. The PF keys also return data. The PA (Program Attention) keys and the CLEAR key (written ***CLR***) do not return data.
<limit>	A constant or scalar in I, N or P format whose value indicates the maximum number of records to be presented.
<logical condition>	See definition in part 1.
<map field name>	Name of a map field prefixed with asterisk.
<map name>	Name of a NATURAL map module to be used in conjunction with an INPUT statement.
<message number>	Reference number for a message stored in the NATURAL message file for the current library or a linked library, prefixed with * (asterisk), optionally followed by a single parameter, enclosed in parentheses, specifying a <field representation attribute> e.g. (AD=I), a <color definition attribute> e.g. (CD=RE), or both e.g. (AD=I, CD=RE).
<non-NATURAL subroutine>	Name of a non-NATURAL subroutine (not the name of the module).
<number of records>	A number of records, specified using a numeric literal.
<parameter assignment>	A string in the format ***XX***=***Y***, where ***XX*** identifies a session parameter and ***Y*** indicates the value to be assigned. Valid session parameters are AD, AL, BX, CC, CD, CF, CV, DC, DU, DY, EJ, EM, ES, FC, FL, FS, GC, HC, HE, HW, IA, IC, ID, IM, IP, IS, KD, LC, LE, LS, LT, MC, MP, MS, MT, NC, NL, PC, PD, PM, PS, SA, SF, SG, SL, SM, TC, UC, WH, ZD, ZP. The assigned value operates at the session, statement or field level, depending on the context.
<program name>	Name of a NATURAL program module.
<repeated character>	A character (***'x'***) to be repeated a number of times (***n***) specified in the format ***'x'(n)***, optionally followed by a single parameter, enclosed in parentheses, specifying a <field representation attribute> e.g. (AD=I), a <color definition attribute> e.g. (CD=RE), or both e.g. (AD=I, CD=RE).
<report number>	Report number in the range 0 to 31, specified using a numeric literal.
scalar	A database field or user-defined variable which is not a constant nor an array range nor a group.

<search criterion>	A relational expression comparing 2 values using one of the following relational operators: = (equal to), EQ (equal to), EQUAL TO, NE (not equal) NOT = (not equal), NOT EQUAL, NOT EQUAL TO, < (less than), LT (less than), LESS THAN, <= (less than or equal to), LE (less than or equal to), LESS EQUAL (less than or equal to), >= (greater than or equal to), GE (greater than or equal to), GREATER EQUAL (greater than or equal to). For all forms of the relational operators EQUAL TO and NOT EQUAL TO, a range may be specified using the keyword THRU, for example COUNTRY = 'AUSTRALIA' THRU 'ZIMBABWE'. A specific value or range of values may be excluded using the keywords BUT NOT, for example COUNTRY = 'AUSTRALIA' THRU 'ZIMBABWE' BUT NOT 'ENGLAND THRU 'GERMANY'. The boolean operators AND, OR and NOT may be used to join multiple relational expressions. Paired parentheses may be used to control the order of evaluation.
<set name>	A named list of ISNs, specified as a <type 2 operand>.
<spacer>	A number (***n***) of blank character positions specified in the format ***n***X.
<starting position>	Starting character position, specified as a <type 7 operand>.
<statement>	A single NATURAL statement.
<u><statement></u>	One or more NATURAL statements.
<subprogram name>	Name of a NATURAL subprogram module, specified as a <type 2 operand>.
<subroutine name>	Name of a NATURAL subroutine (not the name of the module), specified as a <type 1 operand>.
<tab>	Tab position specified in the format ***n***T, where ***n*** is the column number.
<text>	Text, specified using an alpha literal, optionally followed by a single parameter, enclosed in parentheses, specifying a <field representation attribute> e.g. (AD=I), a <color definition attribute> e.g. (CD=RE), or both e.g. (AD=I, CD=RE).
<type 1 operand>	A constant in A format.
<type 2 operand>	A constant or scalar in A format.
<type 3 operand>	A constant or scalar in A format, optionally followed by a single parameter, enclosed in parentheses, specifying a <field representation attribute> e.g. (AD=I), a <color definition attribute> e.g. (CD=RE), or both e.g. (AD=I, CD=RE).
<type 4 operand>	A constant or scalar in A, B, D, F, I, L, N, P, or T format.
<type 5 operand>	A constant or scalar in A, B or N format.
<type 6 operand>	A constant or scalar in B, I, N or P format.
<type 7 operand>	A constant or scalar in I, N or P format.
<type 8 operand>	A constant or scalar in N or P format.
<type 9 operand>	A constant, scalar or array range in A format.
<type 10 operand>	A constant, scalar or array range in A or B format.
<type 11 operand>	A constant, scalar or array range in A, B, C, D, F, I, L, N, P, or T format.
<type 12 operand>	A constant, scalar or array range in A, B, D, F, I, L, N, P, or T format.
<type 13 operand>	A constant, scalar or array range in B, I, N or P format.
<type 14 operand>	A constant, scalar, array range or group in A, B, C, D, F, I, L, N, P, or T format.
<type 15 operand>	A constant, scalar, array range, group or system variable in A format.

<type 16 operand>	A constant, scalar, array range, group or system variable in A, B, D, F, I, N, P, or T format.
<type 17 operand>	A constant, scalar, array range, group or system variable in A, B, D, F, I, L, N, P, or T format.
<type 18 operand>	A constant, scalar, array range or system variable in F, I, N or P format.
<type 19 operand>	A constant, scalar, array range or modifiable system variable in F, I, N or P format.
<type 20 operand>	A constant, scalar, array range or system variable in D, F, I, N, P, or T format.
<type 21 operand>	A constant, scalar, array range or system variable in A, B, C, D, F, I, L, N, P, or T format.
<type 22 operand>	A constant, scalar, array range or system variable in A, B, D, F, I, L, N, P, or T format.
<type 23 operand>	A constant, scalar, array range or system variable in A, B, D, F, I, N, P, or T format.
<type 24 operand>	A constant, scalar or system variable in A, B, D, F, I, N, P, or T format.
<type 25 operand>	A constant, scalar or system variable in B, I, N or P format.
<type 26 operand>	A constant, scalar or system variable in F, I, N or P format.
<type 27 operand>	A scalar in A format.
<type 28 operand>	A scalar in A, B, D, F, I, L, N, P, or T format.
<type 29 operand>	A scalar in A, B, D, F, I, N, P, or T format.
<type 30 operand>	A scalar in A, N, or P format.
<type 31 operand>	A scalar in F, I, N or P format.
<type 32 operand>	A scalar in I, N or P format.
<type 33 operand>	A scalar in N or P format.
<type 34 operand>	A scalar in I4 format.
<type 35 operand>	A scalar or array range in A format.
<type 36 operand>	A scalar or array range in A or B format.
<type 37 operand>	A scalar or array range in A, B, C, D, F, I, L, N, P, or T format.
<type 38 operand>	A scalar or array range in A, B, D, F, I, L, N, P, or T format.
<type 39 operand>	A scalar or array range in A, B, F, I, N or P format.
<type 40 operand>	A scalar or array range in A, C or L format.
<type 41 operand>	A scalar, array range or group in A format.
<type 42 operand>	A scalar, array range or group in A, B, C, D, F, I, L, N, P, or T format.
<type 43 operand>	A scalar, array range or group in A, B, D, F, I, L, N, P, or T format.
<type 44 operand>	A scalar, array range, group or system variable in A, B, D, F, I, L, N, P, or T format optionally followed by a single parameter, enclosed in parentheses, specifying a <field representation attribute> e.g. (AD=I), a <color definition attribute> e.g. (CD=RE), or both e.g. (AD=I, CD=RE).
<type 45 operand>	A scalar, array range or system variable in A, B, D, F, I, L, N, P, or T format
<type 46 operand>	A scalar, array range or modifiable system variable in A, B, C, D, F, I, L, N, P, or T format.
<type 47 operand>	A scalar, array range or modifiable system variable in A, B, D, F, I, N, P, or T format.
<type 48 operand>	A scalar, array range or modifiable system variable in D, F, I, N, P, or T format.
<type 49 operand>	A scalar, array range or modifiable system variable in F, I, N or P format.

<type 50 operand>	A group field.
<type 51 operand>	An array range in A format.
<type 52 operand>	An array range in A2 or B2 format (i.e. A2/1:n or B2/1:n).
<type 53 operand>	A scalar or array range in A, B, N or P format.
<view name>	Name of a database view.
<window name>	A literal string (contained in single quotes) specifying the name of a window which has been defined using a DEFINE WINDOW statement.
<work file number>	The number of a work file (sequential file), specified using a numeric literal.

(page intentionally blank)

Part 3
Essential Logic

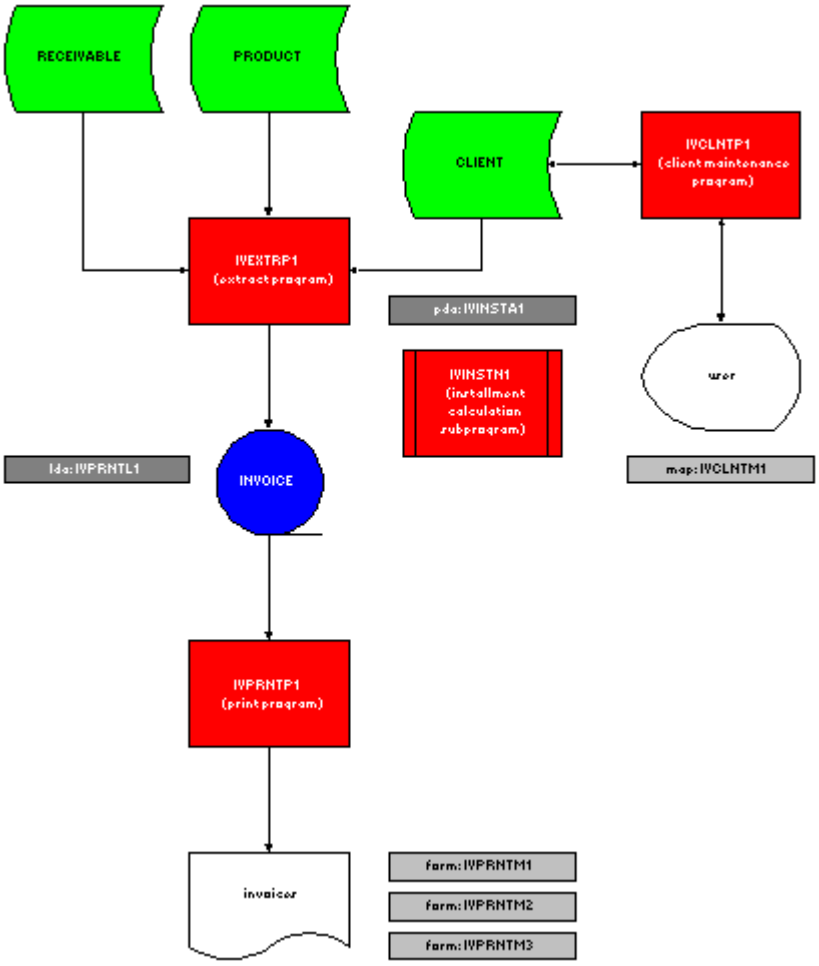
Part 3 - Essential Logic

This part of the course revisits many of the concepts from part 1 and much of the syntax from part 2 and presents them in the context of an application, to illustrate how things fit together. Admittedly, the application is a little contrived, but it serves its purpose without excess baggage.

This part of the course is arranged as follows:

- application overview
- file specifications
- inventory of modules
- IVINSTN1 - specifications, source code and narrative
- IVEXTRP1 - specifications, source code and narrative
- IVPRNTP1 - specifications, source code and narrative
- IVCLNTP1 - specifications, source code and narrative

Application overview



3 database files CLIENT, RECEIVABLE and PRODUCT are accessed by a batch program IVEXTRP1. This program determines which clients are due to be billed and creates a sequential file INVOICE, containing the information necessary to produce an invoice. Subprogram IVINSTN1 is used to do some of the calculations. File INVOICE is read by batch program IVPRNTP1 which prints the invoices. On-line program IVCLNTP1 allows users to maintain database file CLIENT.

File specifications

Database file RECEIVABLE		
Field	**Format & Length**	**Notes**
CLIENT-ID	N8	key
UNPAID-FLAG	L	key
PRODUCT-ID	N6	
QUANTITY	N3	
TOTAL-PRICE	P7.2	
PAID-AMOUNT	P7.2	
C*BILL-DATE		system-maintained counter
BILL-DATE	D/12	multiple field
AUDIT-TIME	T	
AUDIT-USER	A8	
AUDIT-PROGRAM	A8	
AUDIT-COUNTER	P7	

Database file PRODUCT		
Field	**Format & Length**	**Notes**
PRODUCT-ID	N6	key
PRODUCT-NAME	A20	key
UNIT-PRICE	P7.2	
NBR-INSTALLMENTS	P2	
C*SALES-MESSAGE		system-maintained counter
SALES-MESSAGE	A40/5	multiple field
AUDIT-TIME	T	
AUDIT-USER	A8	
AUDIT-PROGRAM	A8	
AUDIT-COUNTER	P7	

Database file CLIENT		
Field	**Format & Length**	**Notes**
CLIENT-ID	N8	key
BILLING-NAME	A40	
C*BILLING-ADDRESS		system-maintained counter
BILLING-ADDRESS	A40/3	multiple field
AUDIT-TIME	T	
AUDIT-USER	A8	
AUDIT-PROGRAM	A8	
AUDIT-COUNTER	P7	

Sequential file INVOICE		
Field	**Format & Length**	**Notes**
CLIENT-ID	N8	
BILLING-NAME	A40	
BILLING-ADDRESS	A40/3	array
PRODUCT-ID	N6	
PRODUCT-NAME	A20	
QUANTITY	N3	
TOTAL-PRICE	N7.2	
NBR-INSTALLMENTS	N2	
DUE-AMOUNT	N7.2/12	array
TOTAL-DUE	N7.2	
DUE-DATE	A8	MMDDYYYY format
SALES-MESSAGE	A40/5	array

Use only alpha and numeric fields in sequential files to simplify debugging and to facilitate access by non-NATURAL systems.

Modules

Inventory of modules			
Library	**Module**	**Type**	**Description**
INVLIB	IVINSTN1	N	Subprogram to calculate installment amounts.
INVLIB	IVINSTA1	A	PDA to define parameters for subprogram IVINSTN1.
INVLIB	IVEXTRP1	P	Batch program to extract data from database file RECEIVABLES.
INVLIB	IVEXTRL1	L	LDA to define sequential file created by program IVEXTRP1.
INVLIB	IVPRNTP1	P	Batch program to print invoices.
INVLIB	IVPRNTM1	M	Form for program IVPRNTP1.
INVLIB	IVPRNTM2	M	Form for program IVPRNTP1.
INVLIB	IVPRNTM3	M	Form for program IVPRNTP1.
INVLIB	IVCLNTP1	P	On-line program to maintain database file CLIENT.
INVLIB	IVCLNTM1	M	Map for program IVCLNTP1.

IVINSTN1 (installment calculation subprogram)

Subprogram to calculate installment amounts from a given total price and a given number of installments.

Specification

TOTAL-PRICE(P7.2) and NBR-INSTALLMENTS(P2) are input parameters.

INSTALLMENT-AMOUNT(P7.2/12) is an output parameter.

Based on TOTAL-PRICE and NBR-INSTALLMENTS, calculate the amount of each installment, truncating to whole. Adjust the last installment to compensate for truncation. This means the final installment may be greater than the others. Return the results in the INSTALLMENT-AMOUNT array.

Source code

```
>                                        > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** SUBPROGRAM IVINSTN1
  0020 ** SUBPROGRAM TO CALCULATE INSTALLMENT AMOUNTS
  0030 **
  0040 ** CREATED ON 01/01/1998 BY SIMPSON
  0050 ************************************************************************
  0060 DEFINE DATA
  0070 PARAMETER USING IVINSTA1 /* PDA
  0080 LOCAL
  0090 1 #FINAL-AMOUNT (P7.2)
  0100 1 #NBR-INSTALLMENTS (P2)
  0110 1 #REG-AMOUNT (P7.2)
  0120 1 #REG-COUNT (P2)
  0130 END-DEFINE
  0140 *
  0150 RESET IVINSTA1.INSTALLMENT-AMOUNT(*)
  0160 *
  0170 #NBR-INSTALLMENTS := IVINSTA1.NBR-INSTALLMENTS
  0180 #REG-AMOUNT := IVINSTA1.TOTAL-PRICE / #NBR-INSTALLMENTS
  0190 #REG-COUNT := #NBR-INSTALLMENTS - 1
  0200 #FINAL-AMOUNT :=
  0210   IVINSTA1.TOTAL-PRICE - (#REG-COUNT * #REG-AMOUNT)
  0220 *
  0230 IVINSTA1.INSTALLMENT-AMOUNT(1:#REG-COUNT) := #REG-AMOUNT
  0240 IVINSTA1.INSTALLMENT-AMOUNT(#NBR-INSTALLMENTS) := #FINAL-AMOUNT
  0250 *
  0260 END /* IVINSTN1
       ....+....1....+....2....+....3....+....4....+....5....+... S 26   L 1
```

Narrative

0010-0050	Comment block. Identifies the module, explains what it does and identifies who wrote it. Also, if applicable, when, why and by whom it was modified.
0060-0130	Data definitions. This module uses both parameter data and local data. The parameter data is defined using PDA IVINSTA1. The local data is defined in-line. The only reason to "externalize" this data (into an LDA) would be to reduce the size of the module if it got too large.
0090-0120	Work variables. Datanames start with # to identify them as such. Arranged alphabetically to make them easier to find. All are defined as format P because they are used in calculations and/or used as an index. Format N would also work, but would be less efficient.
0150-0260	Logic is simple enough that it doesn't need to be split into subfunctions. Consequently, there are no in-line subroutines.
0150	Logic starts with a RESET statement to initialize the output parameters. A good practice for all subprograms.
0170	Assignment statement to initialize #NBR-INSTALLMENTS. Necessary because qualified datanames such as IVINSTA1.NBR-INSTALLMENTS cannot be used as an index. Dataname is identical except for the # prefix.
0180	Assignment statement to calculate regular installment amount and store result in #REG-AMOUNT. The result will be truncated to give whole cents.
0190	Assignment statement to calculate number of regular installments and store result in #REG-COUNT.
0200-0210	Assignment statement to calculate amount of last installment and store result in #FINAL-AMOUNT.
0230-0240	Assignment statements to store calculated values in output parameters. Both statements reference specific elements of the array.
0260	END marks the physical end of the module.

Ancillary source code – IVINSTA1

```
Parameter IVINSTA1  Library INVLIB                          DBID   1 FNR   1
Command                                                              > +
I T L Name                            F Leng Index/Init/EM/Name/Comment
All - ---------------------------------- - ---- ---------------------------------
    1 IVINSTA1
    2 INPUT-PARAMETERS
    3 TOTAL-PRICE                     P  7.2
    3 NBR-INSTALLMENTS                P    2
    2 OUTPUT-PARAMETERS
    3 INSTALLMENT-AMOUNT              P  7.2 (1:12)









------------------------------------------------------------------- S 6    L 1
```

IVEXTRP1 (extract program)

Batch program to determine which clients have payments due, extract all data necessary to generate an invoice and write the data to a sequential file.

Specification

Read RECEIVABLE using UNPAID-FLAG as the key.

If PAID-AMOUNT is greater than or equal to TOTAL-PRICE, ignore the record and begin processing the next.

Call subprogram IVINSTN1 to calculate the amount of each installment. Maximum on any product is 12 installments.

Based on the bill dates and the current date, determine which installments are due or past due, i.e. candidates for billing. Based on the amount of each installment and the total amount paid to date, determine which candidate installments are not fully paid. For each one, calculate the amount due and store the result in an array. The contents of the array will indicate which installments have money due, and how much. Also accumulate the due or past amounts into a separate total.

If the total amount due is greater than zero, write a record to INVOICE. This sequential file will be sorted (using an external utility) and used to produce invoices.

Access CLIENT using CLIENT-ID as the key, to obtain billing name and billing address.

Access PRODUCT using PRODUCT-ID as the key, to obtain the product name and sales message.

Calculate DUE-DATE. Unless an installment is overdue, this will be the current date plus 30 days. If any installment is overdue, it will be the current date plus 7 days.

When all records have been processed, display counts for records read and records written.

Make this program restartable. The external sort will delete any duplicate records created as a result of a restart. Once this program starts, no other applications will update RECEIVABLES until it has completed successfully.

Source code

```
>                                       > +  Program     IVEXTRP1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** PROGRAM IVEXTRP1
  0020 ** BATCH PROGRAM TO EXTRACT DATA FROM DATABASE FILE RECEIVABLES
  0030 ** CONTAINS AUTOMATIC RESTART LOGIC.
  0040 **
  0050 ** CREATED ON 01/01/1998 BY SIMPSON
  0060 ************************************************************
  0070 DEFINE DATA
  0080 LOCAL USING IVINSTA1 /* PDA FOR SUBPROGRAM IVINSTN1
  0090 LOCAL USING IVEXTRL1 /* DEFINES SEQUENTIAL FILE
  0100 LOCAL
  0110 1 #DUE-AMOUNT (P7.2/12)
  0120 1 #DUE-DATE (D)
  0130 1 #ET-COUNT (P3)
  0140 1 #INDEX (P2)
  0150 1 #NBR-INSTALLMENTS (P2)
  0160 1 #PAID-AMOUNT (P7.2)
  0170 *
  0180 1 #RESTART-DATA (A10)
  0190 1 REDEFINE #RESTART-DATA
  0200   2 #READ-COUNT (P8)
  0210   2 #WRITE-COUNT (P8)
  0220 *
  0230 1 #TOTAL-DUE (P7.2)
  0240 *
  0250 1 CLIENT VIEW OF CLIENT
  0260   2 BILLING-NAME
  0270   2 BILLING-ADDRESS (1:3)
  0280 *
  0290 1 PRODUCT VIEW OF PRODUCT
  0300   2 PRODUCT-NAME
  0310   2 SALES-MESSAGE (1:5)
  0320 *
  0330 1 RECEIVABLE VIEW OF RECEIVABLE
  0340   2 CLIENT-ID
  0350   2 PRODUCT-ID
  0360   2 QUANTITY
  0370   2 TOTAL-PRICE
  0380   2 PAID-AMOUNT
  0390   2 C*BILL-DATE
  0400   2 BILL-DATE
  0410   2 UNPAID-FLAG
  0420   2 AUDIT-TIME
  0430   2 AUDIT-PROGRAM
  0440   2 AUDIT-USER
  0450   2 AUDIT-COUNTER
  0460 END-DEFINE
  0470 *
  0480 GET TRANSACTION DATA #RESTART-DATA
  0490 IF #RESTART-DATA = ' '
  0500   RESET
  0510     #RESTART-DATA.#READ-COUNT
  0520     #RESTART-DATA.#WRITE-COUNT
  0530 END-IF
  0540 *
```

```
0550   READ-RECEIVABLE.
0560 READ RECEIVABLE BY UNPAID-FLAG
0570 *
0580   ADD 1 TO #ET-COUNT
0590   IF #ET-COUNT > 200
0600     RESET #ET-COUNT
0610     END TRANSACTION #RESTART-DATA
0620   END-IF
0630 *
0640   IF *COUNTER(READ-RECEIVABLE.) LE #RESTART-DATA.#READ-COUNT
0650     ESCAPE TOP
0660   END-IF
0670 *
0680   ADD 1 TO #RESTART-DATA.#READ-COUNT
0690 *
0700   IF RECEIVABLE.PAID-AMOUNT GE RECEIVABLE.TOTAL-PRICE
0710     ESCAPE TOP
0720   END-IF
0730 *
0740   PERFORM CALC-AMOUNTS-DUE
0750   IF #TOTAL-DUE = 0
0760     ESCAPE TOP
0770   END-IF
0780 *
0790   PERFORM PULL-CLIENT-INFO
0800   PERFORM PULL-PRODUCT-INFO
0810   PERFORM WRITE-INVOICE-REC
0820   ADD 1 TO #RESTART-DATA.#WRITE-COUNT
0830 *
0840 END-READ
0850 *
0860 WRITE ‘ INPUT RECORDS:’ #RESTART-DATA.#READ-COUNT (EM=ZZ,ZZZ,ZZ9)
0870 WRITE ‘OUTPUT RECORDS:’ #RESTART-DATA.#WRITE-COUNT (EM=ZZ,ZZZ,ZZ9)
0880 *
0890 RESET #RESTART-DATA
0900 END TRANSACTION #RESTART-DATA
0910 *
0920 DEFINE SUBROUTINE CALC-AMOUNTS-DUE
0930 ************************************************************
0940 ** Calculates amount(s) due.
0950 ************************************************************
0960 IVINSTA1.TOTAL-PRICE := RECEIVABLE.TOTAL-PRICE
0970 IVINSTA1.NBR-INSTALLMENTS := RECEIVABLE.C*BILL-DATE
0980 CALLNAT 'IVINSTN1' IVINSTA1
0990 *
1000 RESET
1010   #DUE-AMOUNT(*)
1020   #TOTAL-DUE
1030 *
1040 #PAID-AMOUNT := RECEIVABLE.PAID-AMOUNT
1050 FOR #INDEX = 1 TO RECEIVABLE.C*BILL-DATE
1060   DECIDE FOR FIRST CONDITION
1070     WHEN RECEIVABLE.BILL-DATE(#INDEX) > *DATX
1080 *                    /* installment not yet due
```

```
1090       ESCAPE BOTTOM
1100     WHEN IVINSTA1.INSTALLMENT-AMOUNT(#INDEX) > #PAID-AMOUNT
1110 *                    /* installment not fully paid
1120       #DUE-AMOUNT(#INDEX) :=
1130         IVINSTA1.INSTALLMENT-AMOUNT(#INDEX) - #PAID-AMOUNT
1140       ADD #DUE-AMOUNT(#INDEX) TO #TOTAL-DUE
1150       RESET #PAID-AMOUNT
1160     WHEN NONE
1170 *                    /* reduce paid amount by installment amount
1180       SUBTRACT IVINSTA1.INSTALLMENT-AMOUNT(#INDEX)
1190         FROM #PAID-AMOUNT
1200   END-DECIDE
1210 END-FOR
1220 END-SUBROUTINE /* CALC-AMOUNTS-DUE
1230 *
1240 DEFINE SUBROUTINE PULL-CLIENT-INFO
1250 ************************************************************
1260 ** Accesses CLIENT to retrieve billing name and address.
1270 ************************************************************
1280 FIND (1) CLIENT WITH CLIENT-ID = RECEIVABLE.CLIENT-ID
1290   IF NO RECORD FOUND
1300     WRITE 'CLIENT' RECEIVABLE.CLIENT-ID 'NOT ON FILE'
1310     STOP
1320   END-NOREC
1330 END-FIND
1340 END-SUBROUTINE /* PULL-CLIENT-INFO
1350 *
1360 DEFINE SUBROUTINE PULL-PRODUCT-INFO
1370 ************************************************************
1380 ** Accesses PRODUCT to retrieve product name and sales message.
1390 ************************************************************
1400 FIND (1) PRODUCT WITH PRODUCT-ID = RECEIVABLE.PRODUCT-ID
1410   IF NO RECORD FOUND
1420     WRITE 'PRODUCT' RECEIVABLE.PRODUCT-ID 'NOT ON FILE'
1430     STOP
1440   END-NOREC
1450 END-FIND
1460 END-SUBROUTINE /* PULL-PRODUCT-INFO
1470 *
1480 DEFINE SUBROUTINE WRITE-INVOICE-REC
1490 ************************************************************
1500 ** Builds and writes an output record.
1510 ************************************************************
1520 RESET INVOICE
1530 MOVE BY NAME RECEIVABLE TO INVOICE
1540 INVOICE.BILLING-NAME := CLIENT.BILLING-NAME
1550 INVOICE.BILLING-ADDRESS(*) := CLIENT.BILLING-ADDRESS(*)
1560 INVOICE.PRODUCT-NAME := PRODUCT.PRODUCT-NAME
1570 INVOICE.SALES-MESSAGE(*) := PRODUCT.SALES-MESSAGE(*)
1580 INVOICE.NBR-INSTALLMENTS := RECEIVABLE.C#BILL-DATE
1590 INVOICE.DUE-AMOUNT(*) := #DUE-AMOUNT(*)
1600 INVOICE.TOTAL-DUE := #TOTAL-DUE
1610 *
1620 #NBR-INSTALLMENTS := RECEIVABLE.C*BILL-DATE
```

```
1630 IF #TOTAL-DUE >
1640     IVINSTA1.INSTALLMENT-AMOUNT(#NBR-INSTALLMENTS)
1650 *                /* total due greater than largest installment
1660   #DUE-DATE := *DATX + 7 /* overdue, 7 days to pay
1670 ELSE
1680   #DUE-DATE := *DATX + 30 /* not overdue, 30 days to pay
1690 END-IF
1700 MOVE EDITED #DUE-DATE (EM=MMDDYYYY) TO INVOICE.DUE-DATE
1710 *
1720 WRITE WORK FILE 1 INVOICE
1730 END-SUBROUTINE /* WRITE-INVOICE-REC
1740 *
1750 END /* IVEXTRP1
     ....+....1....+....2....+....3....+....4....+....5....+... S 175  L 1
```

Narrative

0010-0050	Comment block. Identifies the module, explains what it does and identifies who wrote it. Also, if applicable, when, why and by whom it was modified.
0070-0460	Data definitions. This module uses only local data. IVINSTA1 is used to define data for IVINSTN1 (subprogram to calculate installments). IVEXTRL1 is used to define the record layout for the output sequential file. All other local data is defined in-line. The only reason to "externalize" this data (into an LDA) would be to reduce the size of the module if it got too large.
0110-0230	Work variables. Datanames start with # to identify them as such. Arranged alphabetically to make them easier to find. Work variables related to database fields have matching names, except sometimes prefixed with #.
0250-0450	Database file definitions. Arranged alphabetically to make them easier to find. Only relevant fields are defined. For repeating fields, only relevant occurrences are defined.
0480-1750	Logic with 4 subfunctions broken out and coded as in-line subroutines. Subroutines are arranged alphabetically to make them easier to find.
0480-0900	Control logic. A READ loop containing calls to the various subroutines, plus restart logic.
0480	GET TRANSACTION DATA to retrieve any restart data from the prior run.
0490-0530	IF statement to determine whether restart data is present. If not, counters are set to zero. This is necessary because #RESTART-DATA is an alpha field containing spaces - the redefined subfields do not contain valid numeric values.
0550-0840	READ loop. All RECEIVABLE records with UNPAID-FLAG set to TRUE will be presented.
0580-0620	For every 200 records read, an END TRANSACTION statement commits prior updates and saves restart data in preparation for a subsequent failure. Following a restart, it's entirely possible that no updates will have occurred when the END TRANSACTION statement executes. Even in this scenario, it still serves a purpose as explained below.
0590	200 is an arbitrary number. There is a DBA controlled limit on both the number of database records which can be read and the number which can be updated in a logical transaction. Executing an END TRANSACTION statement every N-th input record prevents these limits from being exceeded. In this particular example, i.e. no other database accesses, 200 is a very conservative number.
0640-0660	IF statement prevents re-processing of records which were processed prior to the failure.
0680	ADD statement increments count of records processed.

0700-0720 IF statement prevents processing of records representing accounts which are fully paid.

0750-0770 IF statement prevents processing of records representing receivables which are not yet due.

0820 Increments count of sequential file records written.

0860-0870 Outputs counts. Edit masks are used to format the output so that counts are shown with commas. In the event of a restart, the number of records physically written to the sequential file may be greater than the count indicates. Any duplicate records would be eliminated by the external sort.

0890-0900 Clears restart data and saves it using END TRANSACTION, which also commits any uncommitted updates.

0920-1220 Subroutine CALC-AMOUNTS-DUE calculates amounts due and populates the #DUE-AMOUNT array and #TOTAL-DUE accordingly. Calls subprogram IVINSTN1 to calculate individual installment amounts. Results could be stored directly in fields on INVOICE, but that would mean the logic to populate INVOICE would be scattered (i.e. not neatly contained in a single subroutine like WRITE-INVOICE-REC) and therefore less readable and harder to maintain.

1050-1210 FOR loop with 1 iteration per installment. #PAID-AMOUNT is dynamically maintained to reflect how much of the total money paid is still available to be allocated. Unless or until it is used up, no additional money is due from the client.

1060-1200 DECIDE FOR statement with 2 conditions plus a NONE condition. The first condition is an installment date in the future. If true, exits the FOR loop. The second condition is an installment amount greater than the amount still available to be allocated. If true, calculates the due amount, stores it in the relevant occurrence of #DUE-AMOUNT and adds it to #TOTAL-DUE. Sets #PAID-AMOUNT to zero to reflect that all money paid has now been allocated. The NONE condition represents a due installment covered by the money already paid. Action is to reduce #PAID-AMOUNT by the installment amount.

1240-1340 Subroutine PULL-CLIENT-INFO brings in client information. FIND (1) is used to access database file
CLIENT. READ could be used instead, but in this case the syntax for FIND is simpler. If the record is not found, outputs an error message and STOPs. Although logic is coded to handle this condition, it is not really expected to occur. Since this error would indicate a serious problem with the application, stopping execution is appropriate.

1360-1460 Subroutine PULL-PRODUCT-INFO brings in product information. FIND (1) is used to access database file
PRODUCT. READ could be used instead, but in this case the syntax for FIND is simpler. If the record is not found, outputs an error message and STOPs. Although logic is coded to handle this condition, it is not really expected to occur. Since this error would indicate a serious problem with the application, stopping execution is appropriate.

1480-1730 Subroutine WRITE-INVOICE-REC builds and writes a sequential file record. Data is obtained from all 3 database files plus some local variables. #DUE-DATE is calculated.

1520 The group level RESET erases any prior data in the record area.

1620 The assignment statement to initialize #NBR-INSTALLMENTS is necessary because qualified datanames such as RECEIVABLE.C*BILL-DATE cannot be used as an index.

1630-1690 The IF statement determines whether the total due is greater than the final (i.e. largest) installment. If so (indicating an overdue amount from a prior installment), calculates due date by adding 7 to the current date. If no amounts are overdue, calculates due date by adding 30 to the current date. Current date is obtained from system variable *DATX.

1700 MOVE EDITED is used to format the due date on the sequential file. Where practical, it's best to avoid packed fields (including date and time fields) on sequential files.

1750 END marks the physical end of the module.

Ancillary source code – IVINSTA1

```
Parameter IVINSTA1  Library INVLIB                          DBID   1 FNR   1
Command                                                              > +
I T L Name                           F Leng Index/Init/EM/Name/Comment
All - -------------------------------- - ---- --------------------------------
    1 IVINSTA1
    2 INPUT-PARAMETERS
    3 TOTAL-PRICE                    P  7.2
    3 NBR-INSTALLMENTS               P    2
    2 OUTPUT-PARAMETERS
    3 INSTALLMENT-AMOUNT             P  7.2 (1:12)









------------------------------------------------------------------- S 6    L 1
```

Ancillary source code – IVEXTRL1

```
Local     IVEXTRL1  Library INVLIB                          DBID 1   FNR 1
Command                                                              > +
I T L Name                           F Leng Index/Init/EM/Name/Comment
All - -------------------------------- - ---- --------------------------------
    1 INVOICE
    2 CLIENT-ID                      N    8
    2 BILLING-NAME                   A   40
    2 BILLING-ADDRESS                A   40 (1:3)
    2 PRODUCT-ID                     N    6
    2 PRODUCT-NAME                   A   20
    2 QUANTITY                       N    3
    2 TOTAL-PRICE                    N  7.2
    2 NBR-INSTALLMENTS               N    2
    2 DUE-AMOUNT                     N  7.2 (1:12)
    2 TOTAL-DUE                      N  7.2
    2 DUE-DATE                       A    8 /* MMDDYYYY
    2 SALES-MESSAGE                  A   40 (1:5)




------------------------------------------------------------------- S 13   L 1
```

IVPRNTP1 (print program)

Batch program to print invoices, using the sequential file INVOICE as input. One invoice per input record.

Specification

Read sequential file INVOICE, which is sorted into ascending CLIENT-ID sequence.

For each input record, print an invoice. Use map modules to control the format. The invoice layout is as follows:

Invoice Layout

```
JANE DOE
6099 BROADMEADOW
SAN ANTONIO, TX 78240



                                        Total   Installment     Amount
Product  Name                    Qty     Price     Number          Due

 008101  BE A BETTER STUDENT      1      99.95     3 of 4          4.98
                                                   4 of 4         25.01
                                                                 --------
                                      PLEASE PAY THIS AMOUNT      29.99
                                        BY  02/01/1998



              Congratulations on your purchase of BE A
              BETTER STUDENT. This purchase entitles
              you to a 10% discount on all new orders
              from our catalog. Call 1-888-STUDENT
              (1-888-788-3368) today for a copy.
```

Source code

```
>                                        > +  Program     IVPRNTP1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** PROGRAM IVPRNTP1
  0020 ** BATCH PROGRAM TO PRINT INVOICES
  0030 **
  0040 ** CREATED ON 01/01/1998 BY SIMPSON
  0050 ************************************************************************
  0060 DEFINE DATA
  0070 LOCAL USING IVEXTRL1 /* DEFINES SEQUENTIAL FILE
  0080 LOCAL
  0090 1 #FORM
  0100   2 #DUE-AMOUNT (N7.2)
  0110   2 #INSTALLMENT-CAPTION (A8) /* 12 of 12
  0120 *
  0130 1 #INSTALLMENT-NBR (P2)
  0140 END-DEFINE
  0150 *
  0160 READ WORK FILE 1 INVOICE
  0170   NEWPAGE
  0180   WRITE NOTITLE USING FORM 'IVPRNTM1'
  0190 *
  0200   FOR #INSTALLMENT-NBR = 1 TO INVOICE.NBR-INSTALLMENTS
  0210     IF INVOICE.DUE-AMOUNT(#INSTALLMENT-NBR) = 0
  0220       ESCAPE TOP
  0230     END-IF
  0240 *
  0250     #FORM.#DUE-AMOUNT := INVOICE.DUE-AMOUNT(#INSTALLMENT-NBR)
  0260     COMPRESS #INSTALLMENT-NBR 'of' INVOICE.NBR-INSTALLMENTS
  0270       INTO #FORM.#INSTALLMENT-CAPTION
  0280     WRITE NOTITLE USING FORM 'IVPRNTM2'
  0290 *
  0300      RESET
  0310       INVOICE.PRODUCT-ID
  0320       INVOICE.PRODUCT-NAME
  0330         INVOICE.QUANTITY
  0340       INVOICE.TOTAL-PRICE
  0350   END-FOR
  0360 *
  0370   WRITE NOTITLE USING FORM 'IVPRNTM3'
  0380 END-WORK
  0390 *
  0400 END /* IVPRNTP1
       ....+....1....+....2....+....3....+....4....+....5....+... S 40   L 1
```

Narrative

0010-0050	Comment block. Identifies the module, explains what it does and identifies who wrote it. Also, if applicable, when, why and by whom it was modified.
0060-0140	Data definitions. This module uses only local data. IVEXTRL1 is used to define the record layout for the input sequential file. All other local data is defined in-line. The only reason to "externalize" this data (into an LDA) would be to reduce the size of the module if it got too large.
0070	LOCAL USING IVEXTRL1 brings in the same record definition used to create the file, thus eliminating the risk of inconsistencies in the definition.
0090-0130	Work variables. Datanames start with # to identify them as such. Arranged alphabetically to make them easier to find. Work variables related to fields on the sequential file have matching names, except sometimes prefixed with #.
0160-0400	Logic is simple enough that it doesn't need to be split into sub-functions. Consequently, there are no in-line subroutines.
0160-0380	Control logic. Logic to read the input file and print an invoice for each input record. The invoice is divided into 3 parts - heading, body and footing - defined in map modules IVPRNTM1, IVPRNTM2 and IVPRNTM3.
0170-0180	Logic to start a new page and print the invoice heading.
0200-0350	Logic to print one detail line per due installment. Since product id, product name, quantity and total price are printed on the first detail line only, the input fields are RESET after the first detail line has been printed.
0370	Logic to print the invoice footing.
0400	END marks the physical end of the module.

Ancillary source code – IVEXTRL1

```
Local     IVEXTRL1  Library INVLIB                              DBID 1   FNR 1
Command                                                                  > +
I T L Name                            F Leng Index/Init/EM/Name/Comment
All - ---------------------------------- - ---- ---------------------------------
    1 INVOICE
    2 CLIENT-ID                       N    8
    2 BILLING-NAME                    A   40
    2 BILLING-ADDRESS                 A   40 (1:3)
    2 PRODUCT-ID                      N    6
    2 PRODUCT-NAME                    A   20
    2 QUANTITY                        N    3
    2 TOTAL-PRICE                     N  7.2
    2 NBR-INSTALLMENTS                N    2
    2 DUE-AMOUNT                      N  7.2 (1:12)
    2 TOTAL-DUE                       N  7.2
    2 DUE-DATE                        A    8 /* MMDDYYYY
    2 SALES-MESSAGE                   A   40 (1:5)




------------------------------------------------------------------- S 13   L 1
```

Ancillary source code – IVPRNTM1

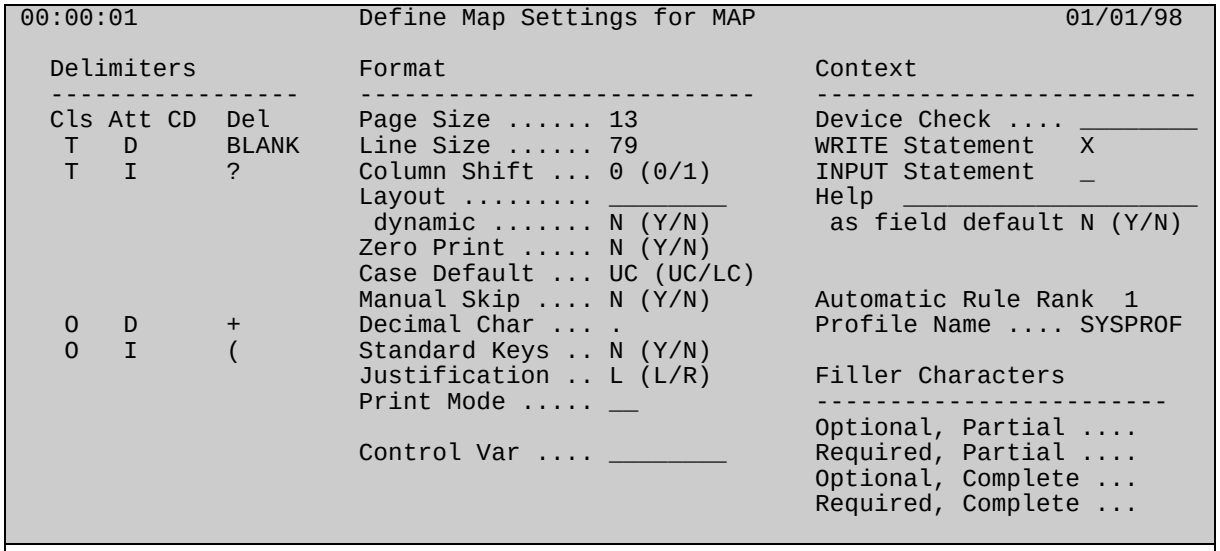
The Map Editor, covered in detail in appendix A, provides a graphical user interface (GUI) allowing you to quickly define screens and reports. It generates code to take care of many tasks, both mundane and complex, which you would otherwise have to code yourself. As with all GUI editors, the generated code can be difficult to read and interpret. Fortunately, it's seldom necessary to do so.

In this part of the course, I've included the summary information provided by the Map Editor, not the generated code, to fill the void between the simple logic of a program like IVPRNTP1 and the results it produces. Since much of the information will be meaningless until you're more familiar with the Map Editor, I've interspersed brief narratives to help you interpret what you see.

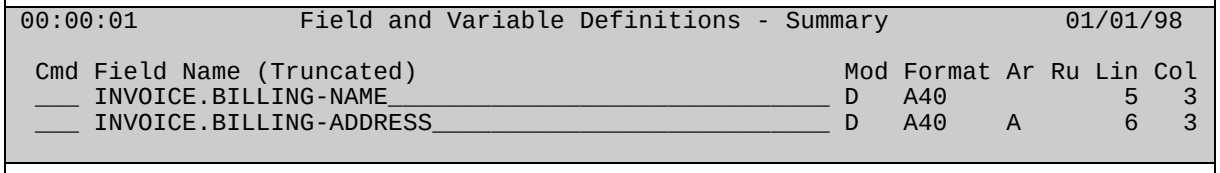
form: IVPRNTM1

```
+XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
+XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
+XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
+XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


                                        Total   Installment     Amount
 Product  Name                  Qty     Price     Number          Due

.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
```

Map module IVPRNTM1 defines the layout for the top part of the invoice. Above is the Map Editor screen graphically showing the layout, including the client name line, the client address lines and the column headings for the invoice detail lines followed by a blank line. The dotted lines at the bottom of the screen indicate unused lines.



Above: Map Editor screen showing the map settings which apply to map module IVPRNTM1. Map attributes, derived from the map profile, can be overridden using this screen. Additionally, installation parameters, which control formatting of data, can be overridden at the map level using this screen. In this case, the default values are used. WRITE is selected, indicating that the map will be used in conjunction with an WRITE USING FORM statement. The page size is defined as 13 lines, meaning that each WRITE statement that utilizes this map will generate 13 lines of output - 4 blank lines, 1 line for client name, 3 lines for client address, 2 blank lines, 2 lines for the column headings and 1 blank line.



Above: Map Editor screen showing the fields defined on the map. Since the fields are attached directly to fields on sequential file INVOICE, data from the current record will be presented automatically - no assignment statements are needed to move the data into the map fields. The map field associated with INVOICE.BILLING-NAME is in A40 format and starts at line 5 column 3. The map field associated with INVOICE.BILLING-ADDRESS is in A40 format and starts at line 6 column 3.

```
Fld INVOICE.BILLING-NAME                                                Fmt A40
 ------------------------------------------------------------------------------
 AD= OD________     ZP=         SG=                                     Rls 0
 AL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY= ____________________________________
 EM= ____________________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to INVOICE.BILLING-NAME. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

```
Arr INVOICE.BILLING-ADDRESS                                             Fmt A40
 ------------------------------------------------------------------------------
 AD= OD________     ZP=         SG=                                     Rls 0
 AL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY= ____________________________________
 EM= ____________________________________________________________________
```

Above: Map Editor screen showing the definition of the map array field attached to INVOICE.BILLING-ADDRESS. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

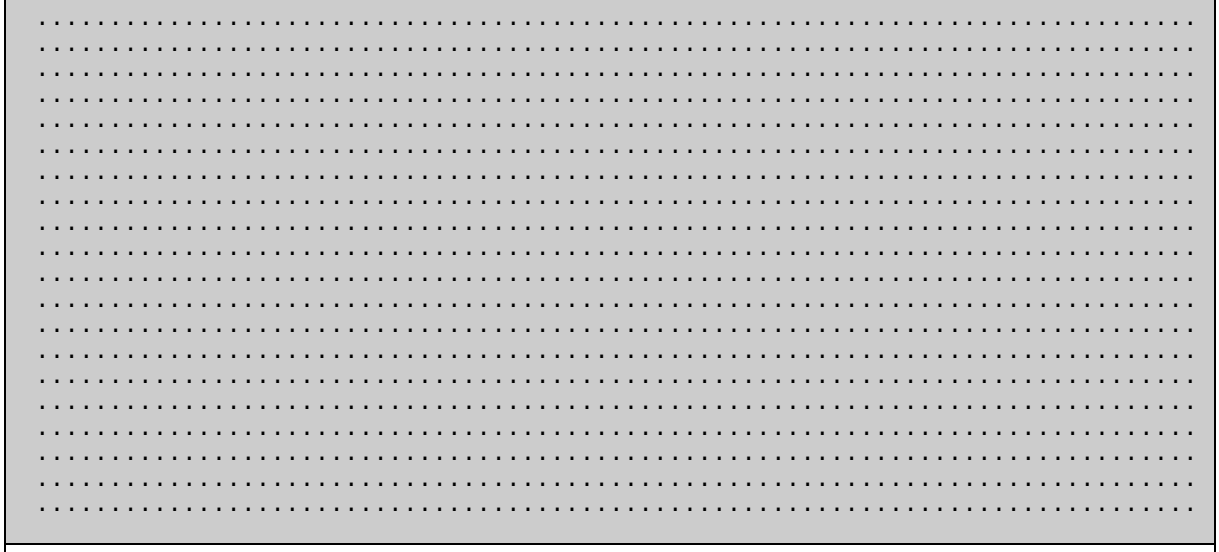
```
Name INVOICE.BILLING-ADDRESS                  Upper Bnds 3_____ 1_____ 1_____
 ------------------------------------------------------------------------------
 Dimensions                    Occurrences    Starting from     Spacing
 1 . Index vertical            3__            1_______          0   Lines
 0 . Index horizontal          1__            ________          1   Columns
 0 . Index (h/v) V             1__            ________          0   Cls/Ls
```

Above: Map Editor screen showing the array definition of the map field attached to INVOICE.BILLING-ADDRESS. This screen allows you to specify which elements of the array are presented, and how they are presented. In this case, no special processing is specified. "Upper Bnds" indicates a 1-dimensional array with 3 occurrences. "Dimensions" indicates the elements of the array will be output vertically. "Occurrences" indicates that 3 occurrences will be output, starting from occurrence 1. "Spacing" indicates that data from the array will be output with 0 blank lines separating each occurrence.

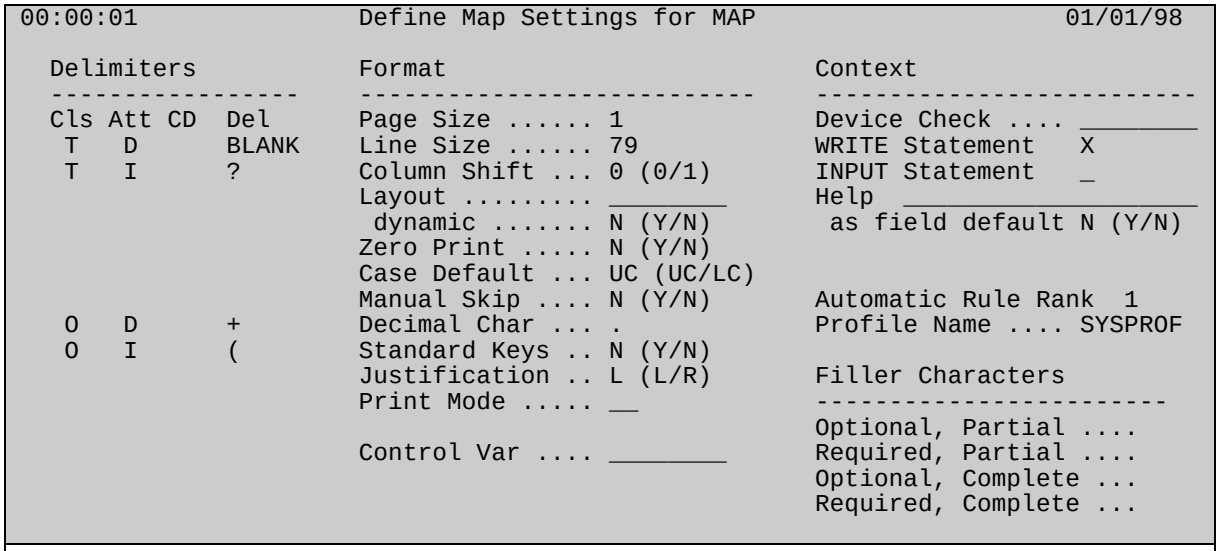
Ancillary source code – IVPRNTM2

form: IVPRNTM2

```
+MMMMMM +XXXXXXXXXXXXXXXXXXX +000 +MMMMMMMMMMM   +XXXXXXXX +MMMMMMMMMMM
```



Map module IVPRNTM2 defines the invoice detail line. Above is the Map Editor screen graphically showing the layout. The dotted lines at the bottom of the screen indicate unused lines.



Above: Map Editor screen showing the map settings which apply to map module IVPRNTM2. Map attributes, derived from the map profile, can be overridden using this screen. Additionally, installation parameters, which control formatting of data, can be overridden at the map level using this screen. In this case, the default values are used. WRITE is selected, indicating that the map will be used in conjunction with an WRITE USING FORM statement. The page size is defined as 1 line, meaning that each WRITE statement that utilizes this map will generate 1 line of output. Unlike the other map modules used, IVPRNTM2 could be called multiple times per invoice.

```
00:00:01          Field and Variable Definitions - Summary          01/01/98

 Cmd Field Name (Truncated)                              Mod Format Ar Ru Lin Col
 ___ INVOICE.PRODUCT-ID_____________________________     D   N6             1   4
 ___ INVOICE.PRODUCT-NAME___________________________     D   A20            1  12
 ___ INVOICE.QUANTITY_______________________________     D   N3             1  34
 ___ INVOICE.TOTAL-PRICE____________________________     D   N7.2           1  39
 ___ #FORM.#INSTALLMENT-CAPTION_____________________     D   A8             1  54
 ___ #FORM.#DUE-AMOUNT______________________________     D   N7.2           1  64
```

Above: Map Editor screen showing the fields defined on the map. Since the fields are attached directly to fields on sequential file INVOICE or work variables, data will be presented automatically - no assignment statements are needed to move the data into the map fields. The map field associated with INVOICE.PRODUCT-ID is in N6 format and starts at line 1 column 4. The map field associated with INVOICE.PRODUCT-NAME is in A20 format and starts at line 1 column 12. The map field associated with INVOICE.QUANTITY is in N3 format and starts at line 1 column 34. The map field associated with INVOICE.TOTAL-PRICE is in N7.2 format and starts at line 1 column 39. The map field associated with #FORM.#INSTALLMENT-CAPTION is in A8 format and starts at line 1 column 54. The map field associated with #FORM.#DUE-AMOUNT is in N7.2 format and starts at line 1 column 64.

```
Fld INVOICE.PRODUCT-ID                                                  Fmt N6
 ------------------------------------------------------------------------------
 AD= OD________     ZP= OFF     SG= OFF                                   Rls 0
 NL= ______         CD= __      CV= ____________________________________  Mod Data
 PM= __             BX= _____   DY= ___________________________________
 EM= 999999_____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to INVOICE-PRODUCT-ID. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. Since this is a numeric field, the ZP (Zero Print) and SG (sign) attributes apply. The ZP attribute (controlling whether a zero value is displayed) defaults to OFF, meaning a zero value will be displayed as spaces. The SG attribute (controlling whether a sign is displayed) defaults to OFF, meaning a sign is not displayed. By specifying an edit mask of 999999, the product id will always be displayed as 6 full digits, i.e. with leading zeros. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

```
Fld INVOICE.PRODUCT-NAME                                                Fmt A20
 ------------------------------------------------------------------------------
 AD= OD________     ZP=          SG=                                    Rls 0
 AL= ______         CD= __       CV= ____________________________________   Mod Data
 PM= __             BX= ____     DY= ____________________________________
 EM= ____________________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to INVOICE.PRODUCT-NAME. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

```
Fld INVOICE.QUANTITY                                                    Fmt N3
 ------------------------------------------------------------------------------
 AD= OD________     ZP= OFF      SG= OFF                                Rls 0
 NL= ______         CD= __       CV= ____________________________________   Mod Data
 PM= __             BX= ____     DY=
 EM= ____________________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to INVOICE-QUANTITY. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. Since this is a numeric field, the ZP (Zero Print) and SG (sign) attributes apply. The ZP attribute (controlling whether a zero value is displayed) defaults to OFF, meaning a zero value will be displayed as spaces. The SG attribute (controlling whether a sign is displayed) defaults to OFF, meaning a sign is not displayed. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

```
Fld INVOICE.TOTAL-PRICE                                                 Fmt N7.2
 ------------------------------------------------------------------------------
 AD= OD________     ZP= OFF      SG= OFF                                Rls 0
 NL= ______         CD= __       CV= ____________________________________   Mod Data
 PM= __             BX= ____     DY=
 EM= ZZZZZZZ9.99_________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to INVOICE-TOTAL-PRICE. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. Since this is a numeric field, the ZP (Zero Print) and SG (sign) attributes apply. The ZP attribute (controlling whether a zero value is displayed) defaults to OFF, meaning a zero value will be displayed as spaces. The SG attribute (controlling whether a sign is displayed) defaults to OFF, meaning a sign is not displayed. By specifying an edit mask of ZZZZZZZ9.99, the total due will be displayed with leading spaces, but at least one digit preceding the decimal point. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

```
Fld #FORM.#INSTALLMENT-CAPTION                                          Fmt A8
 ------------------------------------------------------------------------------
 AD= ODL_______     ZP=         SG=                                     Rls 0
 AL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY= ____________________________________
 EM= ____________________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #FORM.#INSTALLMENT-CAPTION. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value ODL in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity, Left justified.

```
Fld #FORM.#DUE-AMOUNT                                                   Fmt N7.2
 ------------------------------------------------------------------------------
 AD= OD_______      ZP= OFF     SG= OFF                                 Rls 0
 NL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY=
 EM= ZZZZZZZ9.99_________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #FORM.#DUE-AMOUNT. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. Since this is a numeric field, the ZP (Zero Print) and SG (sign) attributes apply. The ZP attribute (controlling whether a zero value is displayed) defaults to OFF, meaning a zero value will be displayed as spaces. The SG attribute (controlling whether a sign is displayed) defaults to OFF, meaning a sign is not displayed. By specifying an edit mask of ZZZZZZZ9.99, the due amount will be displayed with leading spaces, but at least one digit preceding the decimal point.The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

Ancillary source code – IVPRNTM3

form: IVPRNTM3

```
                                                        --------
                                      PLEASE PAY THIS AMOUNT+MMMMMMMMMMM
                                        BY +MMMMMMMMMM



                  +XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                  +XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                  +XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                  +XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                  +XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 ................................................................................
 ................................................................................
 ................................................................................
 ................................................................................
 ................................................................................
 ................................................................................
 ................................................................................
 ................................................................................
 ................................................................................
```

Map module IVPRNTM3 defines the layout for the bottom part of the invoice. Above is the Map Editor screen graphically showing the layout, including a dashed line, the total line, the due date line, some blank lines and the sales message lines. The dotted lines at the bottom of the screen indicate unused lines.

```
00:00:01              Define Map Settings for MAP                    01/01/98

  Delimiters           Format                          Context
  -----------------    ---------------------------     -------------------------
  Cls Att CD  Del      Page Size ...... 11             Device Check .... ________
   T   D      BLANK    Line Size ...... 79             WRITE Statement   X
   T   I      ?        Column Shift ... 0 (0/1)        INPUT Statement   _
                       Layout ......... ________       Help ____________________
                        dynamic ....... N (Y/N)         as field default N (Y/N)
                       Zero Print ..... N (Y/N)
                       Case Default ... UC (UC/LC)
                       Manual Skip .... N (Y/N)        Automatic Rule Rank  1
   O   D      +        Decimal Char ... .              Profile Name .... SYSPROF
   O   I      (        Standard Keys .. N (Y/N)
                       Justification .. L (L/R)        Filler Characters
                       Print Mode ..... __             ----------------------
                                                       Optional, Partial ....
                       Control Var .... ________       Required, Partial ....
                                                       Optional, Complete ...
                                                       Required, Complete ...
```

Above: Map Editor screen showing the map settings which apply to map module IVPRNTM3. Map attributes, derived from the map profile, can be overridden using this screen. Additionally, installation parameters, which control formatting of data, can be overridden at the map level using this screen. In this case, the default values are used. WRITE is selected, indicating that the map will be used in conjunction with an WRITE USING FORM statement. The page size is defined as 11 lines, meaning that each WRITE statement that utilizes this map will generate 11 lines output –a dashed line, the total line, the due date line, 3 blank lines and 5 lines for the sales message. The dotted lines at the bottom of the screen indicate unused lines.

```
00:00:01          Field and Variable Definitions - Summary           01/01/98

 Cmd Field Name (Truncated)                          Mod Format Ar Ru Lin Col
 ___ INVOICE.TOTAL-DUE_____________________________  D   N7.2           2  64
 ___ INVOICE.DUE-DATE______________________________  D   A8              3  47
 ___ INVOICE.SALES-MESSAGE_________________________  D   A40    A        7  16
```

Above: Map Editor screen showing the fields defined on the map. Since the fields are attached directly to fields on sequential file INVOICE, data will be presented automatically - no assignment statements are needed to move the data into the map fields. The map field associated with INVOICE.TOTAL-DUE is in N7.2 format and starts at line 2 column 64. The map field associated with INVOICE.DUE-DATE is in A8 format and starts at line 3 column 47. The map field associated with INVOICE.SALES-MESSAGE is an array field which starts in A40 format which starts at line 7 column 16.

```
Fld INVOICE.TOTAL-DUE                                                    Fmt N7.2
 ------------------------------------------------------------------------------
 AD= OD________     ZP= OFF     SG= OFF                                    Rls 0
 NL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY=
 EM= ZZZZZZZ9.99_____________________________________________________
```

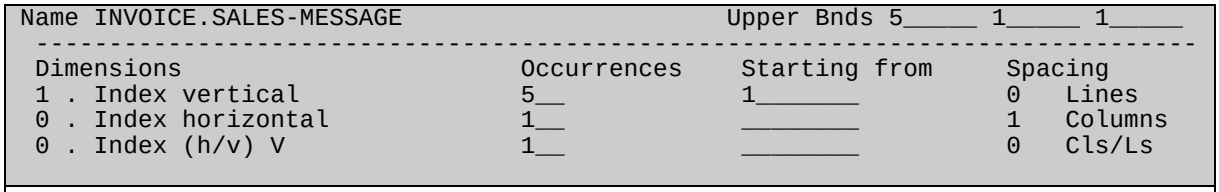
Above: Map Editor screen showing the definition of the map field attached to INVOICE-TOTAL-DUE. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. Since this is a numeric field, the ZP (Zero Print) and SG (sign) attributes apply. The ZP attribute (controlling whether a zero value is displayed) defaults to OFF, meaning a zero value will be displayed as spaces. The SG attribute (controlling whether a sign is displayed) defaults to OFF, meaning a sign is not displayed. By specifying an edit mask of ZZZZZZZ9.99, the total due will be displayed with leading spaces, but at least one digit preceding the decimal point. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.

```
Fld INVOICE.DUE-DATE                                                     Fmt A8
 ------------------------------------------------------------------------------
 AD= ODL_______     ZP=         SG=                                        Rls 0
 AL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY= ___________________________________
 EM= XX'/'XX'/'XXXX_____________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to INVOICE.DUE-DATE. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. By specifying an edit mask of XX'/'XX'/'XX , the date will be displayed with slashes inserted. The value ODL in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity, Left justified.

```
Arr INVOICE.SALES-MESSAGE                                                Fmt A40
 ------------------------------------------------------------------------------
 AD= OD________     ZP=         SG=                                        Rls 0
 AL= ______         CD= __      CV= ____________________________________   Mod Data
 PM= __             BX= ____    DY= ___________________________________
 EM= _____________________________________________________________________
```

Above: Map Editor screen showing the definition of the map array field attached to INVOICE.SALES-MESSAGE. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value OD in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity.



Above: Map Editor screen showing the array definition of the map field attached to INVOICE.SALES-MESSAGE. This screen allows you to specify which elements of the array are presented, and how they are presented. In this case, no special processing is specified. "Upper Bnds" indicates a 1-dimensional array with 5 occurrences. "Dimensions" indicates the elements of the array will be output vertically. "Occurrences" indicates that 5 occurrences will be output, starting from occurrence 1. "Spacing" indicates that data from the array will be output with 0 blank lines separating each occurrence.

IVCLNTP1 (client maintenance program)

On-line program to allow the user to maintain database file CLIENT.

Specification

Allow new records to be added and existing records (if any) to purged or modified, using CLIENT-ID as the key. Allow all of the following actions: A (add), C (clear), D (display), M (modify), N (next), P (purge).

When the action is A (add), allow the user to specify values for all fields except audit data. Populate audit data programmatically.

When the action is C (clear), clear all fields on the screen.

When the action is D (display), force the user to enter a valid client id.

When the action is M (modify), allow all fields to be modified except CLIENT-ID and audit data. Populate audit data programmatically. Disallow the action if the record was changed by another user (or external process) after being displayed.

When the action is N (next), display the record with the next highest client id after the one specified. If no client id is specified, display the record with the lowest client id.

When the action is P (purge), purge the record currently displayed. Disallow the action if the record was changed by another user (or external process) after being displayed.

Source code

```
>                                       > +  Program     IVCLNTP1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** PROGRAM IVCLNTP1
  0020 ** ON-LINE PROGRAM TO MAINTAIN DATABASE FILE CLIENT
  0030 **
  0040 ** CREATED ON 01/01/1998 BY SIMPSON
  0050 **
  0060 ** MODIFIED ON 02/02/1998 BY SIMPSON
  0070 **  CHANGED MESSAGES TO MAKE THEM MORE CONSISTENT
  0080 ************************************************************************
  0090 DEFINE DATA
  0100 LOCAL
  0110 1 #CV-CLIENT-ID (C)
  0120 1 #ISN (P10)
  0130 *
  0140 1 #MAP
  0150   2 #ACTION (A1)
  0160   2 CLIENT-ID (N8)
  0170   2 BILLING-NAME (A40)
  0180   2 BILLING-ADDRESS (A40/1:3)
  0190   2 #AUDIT-TIME (A19)
  0200   2 AUDIT-USER (A8)
  0210   2 AUDIT-PROGRAM (A8)
  0220 *
  0230 1 #MARK (P8)
  0240 1 #MSG (A75)
  0250 1 #TARGET-CLIENT-ID (N8)
  0260 *
  0270 1 DISPLAY-VIEW VIEW OF CLIENT
  0280   2 CLIENT-ID
  0290   2 BILLING-NAME
  0300   2 BILLING-ADDRESS (1:3)
  0310   2 AUDIT-TIME
  0320   2 AUDIT-USER
  0330   2 AUDIT-PROGRAM
  0340   2 AUDIT-COUNTER
  0350 *
  0360 1 UPDATE-VIEW VIEW OF CLIENT
  0370   2 CLIENT-ID
  0380   2 BILLING-NAME
  0390   2 BILLING-ADDRESS (1:3)
  0400   2 AUDIT-TIME
  0410   2 AUDIT-USER
  0420   2 AUDIT-PROGRAM
  0430   2 AUDIT-COUNTER
  0440 END-DEFINE
  0450 *
  0460 SET KEY PF3=PGM
  0470 *
  0480 INPUT USING MAP 'IVCLNTM1'
  0490 IF *PF-KEY = 'PF3'
  0500   STOP
  0510 END-IF
  0520 *
  0530 EXAMINE #MAP.#ACTION TRANSLATE INTO UPPER CASE
  0540 DECIDE ON FIRST VALUE OF #MAP.#ACTION
```

```
0550   VALUE 'A'
0560     PERFORM VALIDATE-KEY
0570     PERFORM ADD-RECORD
0580   VALUE 'C'
0590     PERFORM CLEAR-MAP
0600   VALUE 'D'
0610     PERFORM VALIDATE-KEY
0620     PERFORM DISPLAY-RECORD
0630   VALUE 'M'
0640     PERFORM CHECK-KEY-UNCHANGED
0650     PERFORM MODIFY-RECORD
0660   VALUE 'N'
0670     PERFORM NEXT-RECORD
0680   VALUE 'P'
0690     PERFORM CHECK-KEY-UNCHANGED
0700     PERFORM PURGE-RECORD
0710   NONE
0720     COMPRESS 'Error: invalid action' #MAP.#ACTION INTO #MSG
0730     #MARK := POS(#MAP.#ACTION)
0740     PERFORM REINPUT-DATA
0750 END-DECIDE
0760 *
0770 DEFINE SUBROUTINE ADD-RECORD
0780 ************************************************************************
0790 ** Controls adding of a record.
0800 ************************************************************************
0810   FIND-NBR.
0820 FIND NUMBER DISPLAY-VIEW WITH CLIENT-ID = #MAP.CLIENT-ID
0830 IF *NUMBER(FIND-NBR.) > 0
0840   #MSG := 'Error: record with same key already exists'
0850   #MARK := POS(#MAP.CLIENT-ID)
0860   PERFORM REINPUT-DATA
0870 END-IF
0880 *
0890 RESET UPDATE-VIEW
0900 MOVE BY NAME #MAP TO UPDATE-VIEW
0910 PERFORM POPULATE-AUDIT-FIELDS
0920 *
0930   STORE.
0940 STORE UPDATE-VIEW
0950 END TRANSACTION
0960 #ISN := *ISN(STORE.)
0970 *
0980 #MSG := 'Record added'
0990 #MARK := POS(#MAP.#ACTION)
1000 PERFORM REINPUT-DATA
1010 END-SUBROUTINE /* ADD-RECORD
1020 *
1030 DEFINE SUBROUTINE CHECK-KEY-UNCHANGED
1040 ************************************************************************
1050 ** Disallows action if key is missing or was changed.
1060 ************************************************************************
1070 DECIDE FOR FIRST CONDITION
1080   WHEN #MAP.CLIENT-ID = 0
```

```
1090     #MSG := 'Error: key missing'
1100   WHEN #MAP.CLIENT-ID NE DISPLAY-VIEW.CLIENT-ID
1110     #MSG := 'Error: key modified'
1120   WHEN ANY
1130     #MARK := POS(#MAP.#ACTION)
1140     PERFORM REINPUT-DATA
1150   WHEN NONE
1160     IGNORE
1170 END-DECIDE
1180 END-SUBROUTINE /* CHECK-KEY-UNCHANGED
1190 *
1200 DEFINE SUBROUTINE CHECK-RECORD-UNCHANGED
1210 ************************************************************************
1220 ** Disallows action if record was changed by an external process
1230 ** after being displayed.
1240 ************************************************************************
1250 IF UPDATE-VIEW.AUDIT-COUNTER NE DISPLAY-VIEW.AUDIT-COUNTER
1260   #MSG := 'Record updated by another user, please start over'
1270   #MARK := POS(#MAP.#ACTION)
1280   PERFORM REINPUT-DATA
1290 END-IF
1300 END-SUBROUTINE /* CHECK-RECORD-UNCHANGED
1310 *
1320 DEFINE SUBROUTINE CLEAR-MAP
1330 ************************************************************************
1340 ** Controls clearing of data in map fields.
1350 ************************************************************************
1360 RESET #MAP
1370 *
1380 #MSG := 'Data cleared'
1390 #MARK := POS(#MAP.#ACTION)
1400 PERFORM REINPUT-DATA
1410 END-SUBROUTINE /* CLEAR-MAP
1420 *
1430 DEFINE SUBROUTINE DISPLAY-RECORD
1440 ************************************************************************
1450 ** Controls displaying of a record.
1460 ************************************************************************
1470   FIND.
1480 FIND (1) DISPLAY-VIEW WITH CLIENT-ID = #MAP.CLIENT-ID
1490   IF NO RECORDS FOUND
1500     COMPRESS 'Error: key' #MAP.CLIENT-ID 'not found' INTO #MSG
1510     #MARK := POS(#MAP.CLIENT-ID)
1520     PERFORM REINPUT-DATA
1530   END-NOREC
1540 END-FIND
1550 *
1560 #ISN := *ISN(FIND.)
1570 *
1580 MOVE BY NAME DISPLAY-VIEW TO #MAP
1590 MOVE EDITED DISPLAY-VIEW.AUDIT-TIME(EM=MM/DD/YYYY' 'HH:II:SS)
1600   TO #MAP.#AUDIT-TIME
1610 #MSG := 'Selected record displayed'
1620 #MARK := POS(#MAP.#ACTION)
```

```
1630 PERFORM REINPUT-DATA
1640 END-SUBROUTINE /* DISPLAY-RECORD
1650 *
1660 DEFINE SUBROUTINE MODIFY-RECORD
1670 ************************************************************************
1680 ** Controls modifying a record.
1690 ************************************************************************
1700   GET-FOR-MODIFY.
1710 GET UPDATE-VIEW #ISN
1720 PERFORM CHECK-RECORD-UNCHANGED
1730 *
1740 MOVE BY NAME #MAP TO UPDATE-VIEW
1750 PERFORM POPULATE-AUDIT-FIELDS
1760 UPDATE(GET-FOR-MODIFY.)
1770 END TRANSACTION
1780 *
1790 #MSG := 'Record modified'
1800 #MARK := POS(#MAP.#ACTION)
1810 PERFORM REINPUT-DATA
1820 END-SUBROUTINE /* MODIFY-RECORD
1830 *
1840 DEFINE SUBROUTINE NEXT-RECORD
1850 ************************************************************************
1860 ** Controls displaying of next record. If a client id was entered by
1870 ** the user, the record with that client ID (or if not available,
1880 ** the next highest) will be displayed. If no records have been
1890 ** previously displayed the record with the lowest client id will be
1900 ** displayed.
1910 ************************************************************************
1920 DECIDE FOR FIRST CONDITION
1930   WHEN #CV-CLIENT-ID MODIFIED
1940     #TARGET-CLIENT-ID := #MAP.CLIENT-ID
1950   WHEN #MAP.CLIENT-ID = 0
1960     #TARGET-CLIENT-ID := 1
1970   WHEN NONE
1980     #TARGET-CLIENT-ID := #MAP.CLIENT-ID + 1
1990 END-DECIDE
2000 *
2010   READ.
2020 READ (1) DISPLAY-VIEW BY CLIENT-ID FROM #TARGET-CLIENT-ID
2030 END-READ
2040 *
2050 IF *COUNTER(READ.) = 0
2060   #MSG := 'End of file reached'
2070   #MARK := POS(#MAP.#ACTION)
2080   PERFORM REINPUT-DATA
2090 END-IF
2100 *
2110 #ISN := *ISN(READ.)
2120 *
2130 MOVE BY NAME DISPLAY-VIEW TO #MAP
2140 MOVE EDITED DISPLAY-VIEW.AUDIT-TIME(EM=MM/DD/YYYY' 'HH:II:SS)
2150   TO #MAP.#AUDIT-TIME
2160 #MSG := 'Next record displayed'
```

```
2170 #MARK := POS(#MAP.#ACTION)
2180 PERFORM REINPUT-DATA
2190 END-SUBROUTINE /* NEXT-RECORD
2200 *
2210 DEFINE SUBROUTINE POPULATE-AUDIT-FIELDS
2220 ************************************************************************
2230 ** Updates audit fields on file (UPDATE-VIEW) and map.
2240 ************************************************************************
2250 UPDATE-VIEW.AUDIT-TIME := *TIMX
2260 UPDATE-VIEW.AUDIT-PROGRAM := *PROGRAM
2270 UPDATE-VIEW.AUDIT-USER := *USER
2280 ADD 1 TO UPDATE-VIEW.AUDIT-COUNTER
2290 *
2300 MOVE EDITED UPDATE-VIEW.AUDIT-TIME(EM=MM/DD/YYYY' 'HH:II:SS)
2310   TO #MAP.#AUDIT-TIME
2320 #MAP.AUDIT-PROGRAM := UPDATE-VIEW.AUDIT-PROGRAM
2330 #MAP.AUDIT-USER := UPDATE-VIEW.AUDIT-USER
2340 END-SUBROUTINE /* POPULATE-AUDIT-FIELDS
2350 *
2360 DEFINE SUBROUTINE PURGE-RECORD
2370 ************************************************************************
2380 ** Controls purging of a record.
2390 ************************************************************************
2400   GET-FOR-PURGE.
2410 GET UPDATE-VIEW #ISN
2420 PERFORM CHECK-RECORD-UNCHANGED
2430 *
2440 DELETE(GET-FOR-PURGE.)
2450 END TRANSACTION
2460 *
2470 #MSG := 'Record purged'
2480 #MARK := POS(#MAP.#ACTION)
2490 PERFORM REINPUT-DATA
2500 END-SUBROUTINE /* PURGE-RECORD
2510 *
2520 DEFINE SUBROUTINE REINPUT-DATA
2530 ************************************************************************
2540 ** Executes REINPUT statement to display data and obtain further
2550 ** input.
2560 ************************************************************************
2570 IF (#MARK = POS(#ACTION))
2580     AND (#MAP.#ACTION NE 'N')
2590   RESET #MAP.#ACTION
2600 END-IF
2610 *
2620 IF SUBSTR(#MSG,1,5) = 'Error'
2630   REINPUT FULL WITH TEXT #MSG
2640     MARK #MARK
2650     ALARM
2660 ELSE
2670   REINPUT FULL WITH TEXT #MSG
2680     MARK #MARK
2690 END-IF
2700 END-SUBROUTINE /* REINPUT-DATA
```

```
2710 *
2720 DEFINE SUBROUTINE VALIDATE-KEY
2730 ************************************************************************
2740 ** Validates key value entered by user.
2750 ************************************************************************
2760 IF #MAP.CLIENT-ID LE 0
2770   #MSG := 'Error: client id is invalid'
2780   #MARK := POS(#MAP.CLIENT-ID)
2790   PERFORM REINPUT-DATA
2800 END-IF
2810 END-SUBROUTINE /* VALIDATE-KEY
2820 *
2830 END /* IVCLNTP1
     ....+....1....+....2....+....3....+....4....+....5....+... S 283  L   1
```

Narrative

0010-0080	Comment block. Identifies the module, explains what it does and identifies who wrote it. Also, if applicable, when, why and by whom it was modified.
0090-0440	Data definitions. This module uses only local data which is defined in-line. The only reason to "externalize" this data (into an LDA) would be to reduce the size of the module if it got too large.
0110-0250	Work variables. Datanames start with # to identify them as such. Arranged alphabetically to make them easier to find. Work variables related to database fields have matching names, except sometimes prefixed with #.
0270-0430	Database file definitions. Arranged alphabetically to make them easier to find. Although only 1 file is accessed, 2 different views are used. While not essential, having separate display and update views adds flexibility. Only relevant fields are defined. For repeating fields, only relevant occurrences are defined.
0460-2830	Logic with 11 sub-functions broken out and coded as in-line subroutines. There is 1 subroutine for each possible action plus 5 utility subroutines. Subroutines are arranged alphabetically to make them easier to find.
0460-0750	Control logic. Very simple logic to interact with the user and pass control to the appropriate subroutine, based on the action specified.
0460	SET KEY statement activates the PF3 key. All other PF keys will be inactive.
0480	Although the INPUT statement is not inside a loop of any kind, it can execute multiple times, control being passed to it via REINPUT statements.
0490-0510	IF statement to detect when PF3 is pressed and STOP execution. Standard PF3 functionality.
0530	EXAMINE TRANSLATE statement converts user input to upper case. Without this, the immediately following DECIDE statement would need to allow for both lower and upper case data from the user. This could also be done at the map level, but would affect all fields, not just action.
0540-0750	DECIDE statement with one entry per valid action and one entry for an invalid action. Neatly documents the valid actions. The action codes and action names used in this program (e.g. A for add, M for modify, P for purge, etc.) are commonly used at many NATURAL sites.

0770-1010 Subroutine ADD-RECORD is called when the user specifies an action of A (add) and the specified key value is valid. Uses FIND NUMBER to determine whether the key already exists. HISTOGRAM could be used instead, but FIND NUMBER is simpler and more efficient. If key already exists, sets-up an error message and calls subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned. If the key does not already exist, continues processing. Uses MOVE BY NAME to save the user-entered data. Calls subroutine POPULATE-AUDIT-FIELDS to update the audit fields on both the record and the map. STOREs the record and commits using END TRANSACTION. Saves the ISN to facilitate a subsequent modify or purge action. Sets-up a confirmation message and calls subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned.

1030-1180 Utility subroutine CHECK-KEY-UNCHANGED determines whether client id was changed by the user after being displayed. This is done using control variable #CV-CLIENT-ID which is attached to the map field. If client id was changed (or is null), disallows action by setting-up an error message and calling subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned. Since this logic needs to be executed for several actions (modify and purge) coding it as a subroutine avoids redundant code.

1200-1300 Utility subroutine CHECK-RECORD-UNCHANGED determines whether the current record was modified by another user (or external process) after being displayed. The field AUDIT-COUNTER exists for this purpose. Rather than checking every field on the record looking for a change, it is assumed that all update processes increment AUDIT-COUNTER. If the record was changed, disallows action by setting-up an error message and calling subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned. Since this logic needs to be executed for several actions (modify and purge) coding it as a subroutine avoids redundant code.

1320-1410 Subroutine CLEAR-MAP is called when the user specifies an action of C (clear). Resets map fields using a group level RESET, sets-up a confirmation message and calls subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned.

1430-1640 Subroutine DISPLAY-RECORD is called when the user specifies an action of D (display) and the specified key value is valid. Uses FIND (1) to access the record. READ could be used instead, but in this case the syntax for FIND is simpler. If the record is not found, sets-up an error message and calls REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned. If the record is found, continues processing. Saves the ISN to facilitate a subsequent modify or purge action. Uses MOVE BY NAME to populate the map fields. Uses MOVE EDITED to format AUDIT-TIME into a readable format. Sets-up a confirmation message and calls REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned.

1580-1600 The MOVE BY NAME and MOVE EDITED statements represent redundant code, being identical to those in subroutine NEXT-RECORD. Sometimes a little redundancy is acceptable, especially if eliminating it would complicate your code or decrease readability disproportionately. In this case, there's no real benefit to coding the 2 statements as a subroutine.

1660-1820 Subroutine MODIFY-RECORD is called when the user specifies an action of M (modify) provided the client id has not been changed since the record was displayed. Uses GET and the stored ISN to bring in the record for update. Calls CHECK-RECORD-UNCHANGED to determine whether the record has been changed by another user (or external process) since being displayed. Control will only be returned if record is unchanged. Uses MOVE BY NAME to save the user-entered data. Calls subroutine POPULATE-AUDIT-FIELDS to update the audit fields on both the record and the map. UPDATEs the record and commits using END TRANSACTION. Sets-up a confirmation message and calls subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned.

1840-2190 Subroutine NEXT-RECORD is called when the user specifies an action of N (next). If a client id was entered by the user, sets #TARGET-CLIENT-ID to that value. If no value was entered by the user and no client id is being displayed, sets #TARGET-CLIENT-ID to 1. Otherwise, sets #TARGET-CLIENT-ID to the currently displayed client ID plus 1. Uses READ (1) to access the target record; since multiple key values are involved, a FIND statement would not work. If no records are read, indicating an end-of-file condition, sets-up a message and calls REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned. Unless an end of file condition occurs, continues processing. Saves the ISN to facilitate a subsequent modify or purge action. Uses MOVE BY NAME to populate the map fields. Uses MOVE EDITED to format AUDIT-TIME into a readable format. Sets-up a confirmation message and calls REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned.

2130-2150 The MOVE BY NAME and MOVE EDITED statements represent redundant code, being identical to those in subroutine DISPLAY-RECORD. Sometimes a little redundancy is acceptable, especially if eliminating it would complicate your code or decrease readability disproportionately. In this case, there's no real benefit to coding the 2 statements as a subroutine.

2210-2340 Utility subroutine POPULATE-AUDIT-FIELDS populates audit fields within UPDATE-VIEW using data from system variables. Also updates corresponding map fields to reflect what was written to the database. Since dynamic system variables like *TIMX have the potential to return different values every time they're called, map data is taken from UPDATE-VIEW, not from system variables. Uses MOVE EDITED to format AUDIT-TIME into a readable format. Since this logic needs to be executed for several actions (modify and purge) coding it as a subroutine avoids redundant code.

2360-2500 Subroutine PURGE-RECORD is called when the user specifies an action of P (purge) provided the client id has not been changed since the record was displayed. Uses GET and the stored ISN to bring in the record to be purged. Calls CHECK-RECORD-UNCHANGED to determine whether the record has been changed by another user (or external process) since being displayed. Control will only be returned if record is unchanged. DELETEs the record and commits using END TRANSACTION. Sets-up a confirmation message and calls subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned. Since the map fields still contain data from the purged record, the user has the option of adding the record back with the same or a different client id.

2520-2700 Utility subroutine REINPUT-DATA executes a REINPUT statement to pass control to the INPUT statement on line 0480. On entry, variable #MARK will indicate the required cursor position - either the action field or the client id field. If the cursor is being positioned at the action field, RESETs the previous action unless it was N (next). Retaining an action code of N (next) allows the user to browse the file simply by pressing ENTER repeatedly. A message of some kind will be output. For messages starting with the word “Error”, the alarm will be sounded. Since the alarm is controlled by including or excluding an ALARM clause, 2 different REINPUT statements are necessary - one with and one without an ALARM clause. Because of the way REINPUT works, control will not be returned to the calling module.

2720-2810 Utility subroutine VALIDATE-KEY ensures a valid client id was entered. Since map field client id is attached to the variable #MAP.CLIENT-ID which is defined as N8, only numeric characters can be entered; the only invalid data that could be entered would be a zero or negative client id. If the client id is invalid, disallows action by setting-up an error message and calling subroutine REINPUT-DATA to output it. Because REINPUT-DATA executes a REINPUT statement, control is not returned.

2830 END marks the physical end of the module.

Ancillary source code – IVCLNTM1

map: IVCLNTM1

```
                    ?Client?Information?Screen



            Action:&X (A, C, D, M, N, P)

         Client ID:&99999999

      Billing Name:&XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

   Billing Address:&XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   &XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   &XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

      Last Updated:+XXXXXXXXXXXXXXXXXX

              User:&XXXXXXXX

           Program:&XXXXXXXX
```

Map module IVCLNTM1 defines the client information screen. Above is the Map Editor screen graphically showing the layout.

```
00:00:01              Define Map Settings for MAP                    01/01/98

  Delimiters           Format                          Context
  -----------------    --------------------------      -------------------------
  Cls Att CD  Del      Page Size ...... 23             Device Check .... ________
   T   D      BLANK    Line Size ...... 79             WRITE Statement   _
   T   I      ?        Column Shift ... 0 (0/1)        INPUT Statement   X
   A   D      _        Layout ......... ________       Help ____________________
   A   I      )         dynamic ....... N (Y/N)         as field default N (Y/N)
   A   N      ¬        Zero Print ..... N (Y/N)
   M   D      &        Case Default ... UC (UC/LC)
   M   I      :        Manual Skip .... N (Y/N)        Automatic Rule Rank  1
   O   D      +        Decimal Char ... .              Profile Name .... SYSPROF
   O   I      (        Standard Keys .. N (Y/N)
                       Justification .. L (L/R)        Filler Characters
                       Print Mode ..... __             ----------------------
                                                       Optional, Partial ....
                       Control Var .... ________       Required, Partial ....
                                                       Optional, Complete ...
                                                       Required, Complete ...
```

Above: Map Editor screen showing the map settings which apply to map module IVCLNTM1. Map attributes, derived from the map profile, can be overridden using this screen. Additionally, installation parameters, which control formatting of data, can be overridden at the map level using this screen. In this case, the default values are used. INPUT is selected, indicating that the map will be used in conjunction with an INPUT USING MAP statement. The page size is defined as 23 lines - the entire screen.

```
00:00:01          Field and Variable Definitions - Summary            01/01/98

 Cmd Field Name (Truncated)                                 Mod Format Ar Ru Lin Col
 ___ #MAP.#ACTION_____________________________________      D   A1             6  29
 ___ #MAP.CLIENT-ID___________________________________      D   N8             8  29
 ___ #MAP.BILLING-NAME________________________________      D   A40           10  29
 ___ #MAP.BILLING-ADDRESS_____________________________      D   A40    A      12  29
 ___ #MAP.#AUDIT-TIME_________________________________      U   A19           16  29
 ___ #MAP.AUDIT-USER__________________________________      D   A8            18  29
 ___ #MAP.AUDIT-PROGRAM_______________________________      D   A8            20  29
```

Above: Map Editor screen showing the fields defined on the map. Since the fields are attached directly to work variables, data will be presented automatically - no assignment statements are needed to move the data into the map fields. The map field associated with #MAP.#ACTION is in A1 format and starts at line 6 column 29. The map field associated with #MAP.CLIENT-ID is in N8 format and starts at line 8 column 29. The map field associated with #MAP.BILLING-NAME is in A40 format and starts at line 10 column 29. The map field associated with #MAP.BILLING-ADDRESS is an array field in A40 format which starts at line 12 column 29. The map field associated with #MAP.#AUDIT-TIME is in A19 format and starts at line 16 column 29. The map field associated with #MAP.AUDIT-USER is in A8 format and starts at line 18 column 29. The map field associated with #MAP.AUDIT-PROGRAM is in A8 format and starts at line 20 column 29.

```
Fld #MAP.#ACTION                                                        Fmt A1
 ------------------------------------------------------------------------------
 AD= MDT'_'____     ZP=          SG=        HE= _____________________   Rls 0
 AL= ______         CD= __       CV= _____________________________   Mod Data
 PM= __             BX= ____     DY= _____________________________
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #MAP.#ACTION. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value MDT'_' in the Attribute Definition (AD) parameter indicates that the field is Modifiable, Default intensity, Translated to upper case, filled with _ (underscore) characters.

```
Fld #MAP.CLIENT-ID                                                      Fmt N8
 ------------------------------------------------------------------------------
 AD= MYLT______     ZP= OFF      SG= OFF    HE= _____________________   Rls 0
 NL= ______         CD= __       CV= #CV-CLIENT-ID_____________   Mod Data
 PM= __             BX= ____     DY=
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #MAP.CLIENT-ID. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used except for the CV (Control Variable) parameter. Assigning control variable #CV-CLIENT-ID to the field will allow it to be interrogated to determine whether field #MAP.CLIENT-ID was modified by the user. The value MYLT'_' in the Attribute Definition (AD) parameter indicates that the field is Modifiable, dYnamic (has a control variable assigned), Left justified, Translated to upper case, filled with _ (underscore) characters.

```
Fld #MAP.BILLING-NAME                                                   Fmt A40
 ------------------------------------------------------------------------------
 AD= MDLT'_'___     ZP=          SG=        HE= _____________________   Rls 0
 AL= ______         CD= __       CV= _____________________________   Mod Data
 PM= __             BX= ____     DY= _____________________________
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #MAP.BILLING-NAME. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value MDLT'_' in the Attribute Definition (AD) parameter indicates that the field is Modifiable, Default intensity, Left justified, Translated to upper case, filled with _ (underscore) characters.

```
Arr #MAP.BILLING-ADDRESS                                              Fmt A40
 ------------------------------------------------------------------------------
 AD= MDLT'_'___     ZP=         SG=        HE= _____________________   Rls 0
 AL= _____          CD= __      CV= _____________________________   Mod Data
 PM= __             BX= ____    DY= _____________________________
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map array field attached to #MAP.BILLING-ADDRESS. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value MDLT'_' in the Attribute Definition (AD) parameter indicates that the field is Modifiable, Default intensity, Left justified, Translated to upper case, filled with _ (underscore) characters.

```
Name #MAP.BILLING-ADDRESS                    Upper Bnds 3_____ 1_____ 1_____
 ------------------------------------------------------------------------------
 Dimensions                    Occurrences    Starting from     Spacing
 1 . Index vertical            3__            1_______          0   Lines
 0 . Index horizontal          1__            ________          1   Columns
 0 . Index (h/v) V             1__            ________          0   Cls/Ls
```

Above: Map Editor screen showing the array definition of the map field attached to #MAP.BILLING-ADDRESS. This screen allows you to specify which elements of the array are presented, and how they are presented. In this case, no special processing is specified. "Upper Bnds" indicates a 1-dimensional array with 3 occurrences. "Dimensions" indicates the elements of the array will be output vertically. "Occurrences" indicates that 3 occurrences will be output, starting from occurrence 1. "Spacing" indicates that data from the array will be output with 0 blank lines separating each occurrence.

```
Fld #MAP.#AUDIT-TIME                                                  Fmt A19
 ------------------------------------------------------------------------------
 AD= ODL_______     ZP=         SG=        HE= _____________________   Rls 0
 AL= _____          CD= __      CV= _____________________________   Mod User
 PM= __             BX= ____    DY= _____________________________
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #MAP.#AUDIT-TIME. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value ODL in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity, Left justified.

```
Fld #MAP.AUDIT-USER                                                        Fmt A8
 ------------------------------------------------------------------------------
 AD= ODL_______     ZP=          SG=         HE= _____________________    Rls 0
 AL= ______         CD= __       CV= _____________________________    Mod Data
 PM= __             BX= ____     DY= _____________________________
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #MAP.#AUDIT-USER. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value ODL in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity, Left justified.

```
Fld #MAP.AUDIT-PROGRAM                                                     Fmt A8
 ------------------------------------------------------------------------------
 AD= ODL_______     ZP=          SG=         HE= _____________________    Rls 0
 AL= ______         CD= __       CV= _____________________________    Mod Data
 PM= __             BX= ____     DY= _____________________________
 EM= ____________________________________________________________
```

Above: Map Editor screen showing the definition of the map field attached to #MAP.#AUDIT-PROGRAM. Installation parameters, which control formatting of data, can be overridden at the field level using this screen. In this case, the default values are used. The value ODL in the Attribute Definition (AD) parameter indicates that the field is Output, Default intensity, Left justified.

One important aspect of NATURAL programming that was touched on in all of the above narratives is naming conventions. Using consistent and meaningful names greatly enhances the readability and maintainability of code. Appendix B contains some guidelines.

When you're ready to start coding, please read appendix A. As the title *Tools & Commands* implies, it contains information about the NATURAL environment and detailed descriptions of the various editors and the essential commands for using the tools.

Appendix A
Essential Tools & Commands

Appendix A – Essential Tools & Commands

This appendix covers some of the tools that are supplied with NATURAL and some of the associated commands. Refer to this appendix when you are ready to start coding. Please note there are other tools, such as a cross-reference utility and an interactive debugger, that are beyond the scope of this course.

This appendix is arranged as follows:

- NATURAL environment
- NATURAL Program Editor
- NATURAL Data Area Editor
- NATURAL Map Editor

NATURAL Environment

The NATURAL environment includes a set of integrated tools. In this course, we'll explore those tools that allow you to create/maintain/compile source code, scan/replace text within source code and execute object code. In addition to the 3 editors, each of which is covered in detail in the sections which follow, the following are also covered in this section of the course:

- development facilities menu
- command line

Development facilities menu

After starting a NATURAL session and selecting development facilities, you may be presented with a menu that looks like this:

```
00:00:00    ***  N A T U R A L  APPLICATION DEVELOPMENT SYSTEM  ***    01/01/98
User XXX999            -  Development Facilities  -          Library INVLIB
                                                             Mode Structured
                                                           Program
                  Code  Function
                  ----  --------------------------
                   C    Create Object
                   E    Edit Object
                   R    Rename Object
                   D    Delete Object
                   X    Execute Program
                   T    DB Command Log Facility
                   B    Debugging Facility
                   L    List Objects, X-Ref
                   G    Global Environment
                   ?    Development Facilities Help
                   .    Exit Development Facilities
                  ----  --------------------------
           Code .. _    Type .. _    Name .. ________________________________


Command ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help  Menu  Exit  C     E     R     D     X     T     L     G     Canc
```

You can select a different library to work in by overtyping the library name in the top right corner of the screen.

Rather than working from the development facilities menu, many programmers prefer to work from the NATURAL command line at the bottom of the screen.

Command line

The NATURAL command line may appear as illustrated above, or may take the form of a “NEXT prompt” - the word NEXT followed by an input field. Although this appendix deals mostly with the on-line NATURAL environment, it’s important to remember that there’s also a batch NATURAL environment and the command line applies to both. In batch mode, commands are given via a dataset.

Many different commands can be executed from the command line. The essential ones for creating, maintaining and executing NATURAL code are as follows:

- LOGON
- SCAN
- EXECUTE
- DELETE
- FINISH
- EDIT

LOGON command

The LOGON command selects a library and makes it current. You can only create or maintain source code in the current library. You may be able to execute modules that don’t reside in the current library, but only if the libraries are linked to facilitate this. Consider the following example:

```
LOGON INVLIB
```

In this example, library INVLIB is made current.

If the selected library does not pre-exist, NATURAL will create a new library with the specified name.

SCAN command

When entered directly into the command line, the SCAN command searches all or selected modules in the current library for a specified string. Consider the following example:

SCAN

Scan criteria are entered on a control screen which opens automatically. Optionally, a replace string can be specified. The control screen is as follows:

```
00:00:01           ***** NATURAL OBJECT MAINTENANCE *****              98-01-01
User XXX999                     - SCAN Main Menu -              Library INVLIB


                        Code  Function
                        ----  --------------------------
                          S   SCAN objects in a Library
                          ?   Help
                          .   Exit
                        ----  --------------------------

       Code ............ _
       Scan value ...... ____________________________________
       Replace value ... ____________________________________
       Library ......... INVLIB__
       Object Name ..... ________
       Object Type ..... ___________
       Absolute Scan ... N                          Trace ... N



Command ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help  Menu  Exit
```

The screen below shows an example of scan criteria:

```
00:00:01           ***** NATURAL OBJECT MAINTENANCE *****              98-01-01
User XXX999                     - SCAN Main Menu -                Library INVLIB


                          Code  Function
                          ----  -------------------------
                            S   SCAN objects in a Library
                            ?   Help
                            .   Exit
                          ----  -------------------------

          Code ............ S
          Scan value ...... CLIENT-ID_____________________
          Replace value ... ________________________________
          Library ......... INVLIB___
          Object Name ..... ___
          Object Type ..... ___________
          Absolute Scan ... Y                          Trace ... Y


No objects found for specified 'scan' criteria.
Command ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help  Menu  Exit
```

The screen below shows the result of a successful scan which found the string in an LDA:

```
<                                  >  Local        IVEXTRL1   Lib INVLIB
Scan CLIENT-ID                           Repl
 I T L Name                               F Leng  Index/Init/EM/Name/Comment
 - - - ------------- ALL -------------- - ---- ------------------------------
     2 CLIENT-ID                          N  8.0
```

The screen below shows the result of a successful scan which found the string in a program:

```
>                                       > + Program     : IVCLNTP1 LIB: INVLIB
SCAN: CLIENT-ID                         REP:
       ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0110 1 #CV-CLIENT-ID (C)
  0160   2 CLIENT-ID (N8)
  0250 1 #TARGET-CLIENT-ID (N8)
  0280   2 CLIENT-ID
  0370   2 CLIENT-ID
  0820 FIND NUMBER DISPLAY-VIEW WITH CLIENT-ID = #MAP.CLIENT-ID
  0850   #MARK := POS(#MAP.CLIENT-ID)
  1080   WHEN #MAP.CLIENT-ID = 0
  1100   WHEN #MAP.CLIENT-ID NE DISPLAY-VIEW.CLIENT-ID
  1480 FIND (1) DISPLAY-VIEW WITH CLIENT-ID = #MAP.CLIENT-ID
  1500     COMPRESS 'Error: key' #MAP.CLIENT-ID 'not found' INTO #MSG
  1510     #MARK := POS(#MAP.CLIENT-ID)
  1930   WHEN #CV-CLIENT-ID MODIFIED
  1940     #TARGET-CLIENT-ID := #MAP.CLIENT-ID
  1950   WHEN #MAP.CLIENT-ID = 0
  1960     #TARGET-CLIENT-ID := 1
  1980     #TARGET-CLIENT-ID := #MAP.CLIENT-ID + 1
  2020 READ (1) DISPLAY-VIEW BY CLIENT-ID FROM #TARGET-CLIENT-ID
  2760 IF #MAP.CLIENT-ID LE 0
       ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
```

The SCAN command entered directly into the command line, as described here, should not be confused with the SC (scan/replace) command available within the various source editors to scan within the module being edited.

EXECUTE command

The EXECUTE command executes a program. It is available in both on-line mode and batch mode. Consider the following examples:

Example 1: Explicit use of EXECUTE

```
EXECUTE IVCLNTP1
```

In this example, program IVCLNTP1 is executed. An object module for the specified program must exist in the current library or a linked library.

Example 2: Implicit use of EXECUTE

```
IVCLNTP1
```

In this example (program name as a command), program IVCLNTP1 is executed. EXECUTE is implicit.

Programs which use sequential files cannot be executed on-line.

Programs which were designed for batch execution, even those that don't use sequential files, should generally not be executed on-line. This is because extensive database accesses have the potential to lock-out legitimate on-line users.

DELETE command

The DELETE command brings up a list of modules in the current library. You can enter a code against those modules to be deleted. Enter a code of S to delete the source module. Enter a code of O to delete the object module. Enter a code of B to delete both source and object module.

Example 1: DELETE without parameters

```
DELETE
```

In this example, all modules in the current library are listed. The output might be as follows:

```
00:00:01                  ***** N A T U R A L *****                  01/01/98
Library INVLIB                      - Delete -                DBID    1  FNR   1

 Number  Src/Obj   Type  M  Message      Number  Src/Obj   Type  M  Message
 ------  --------  ----  -  -----------  ------  --------  ----  -  ----------
   01    IVCLNTM1  S/O   _                02    IVCLNTP1  S/O   _
   03    IVEXTRP1  S/O   _                04    IVINSTA1  S/O   _
   05    IVINSTN1  S/O   _                06    IVEXTRL1  S/O   _
   07    IVPRNTM1  S/O   _                08    IVPRNTM2  S/O   _
   09    IVPRNTM3  S/O   _                10    IVPRNTP1  S/O   _









 Mark objects with S (delete source) or O (delete object) or B (delete both),
 or ENTER . to exit  _
```

Example 2: DELETE with parameters

```
DELETE IVPRNT*
```

In this example, all modules with names beginning “IVPRNT” are listed. The output might be as follows:

```
00:00:01                  ***** N A T U R A L *****                  01/01/98
Library INVLIB                      - Delete -                DBID    1  FNR   1

 Number  Src/Obj   Type  M  Message      Number  Src/Obj   Type  M  Message
 ------  --------  ----  -  -----------  ------  --------  ----  -  ----------
   01    IVEXTRL1  S/O   _                 02    IVPRNTM1  S/O   _
   03    IVPRNTM2  S/O   _                 04    IVPRNTM3  S/O   _
   05    IVPRNTP1  S/O   _                 10




 Mark objects with S (delete source) or O (delete object) or B (delete both),
 or ENTER . to exit  _
```

FINISH command

The FINISH command terminates a NATURAL session. The syntax is as follows:

```
F
```

EDIT command

The EDIT command makes pre-existing source available for editing or invokes an editor with an empty file. Source code being edited resides in a section of memory known as the “edit buffer”.

As explained in part 1, there are 3 different specialized editors. To recap, these are as follows:

- Data Area Editor (for LDAs, PDAs and GDAs)
- Program Editor (for programs, subprograms, subroutines, helproutines and copycode)
- Map Editor (for maps)

All 3 editors are described in the sections which follow.

When editing a pre-existing module, NATURAL automatically invokes the appropriate editor, as illustrated in the following examples:

Example 1: edit a pre-existing PDA

```
E IVINSTA1
```

In this example, the Data Area Editor is invoked, since the module to be edited is a PDA.

Example 2: edit a pre-existing program

```
E IVPRNTP1
```

In this example, the Program Editor is invoked since the module to be edited is a program.

When creating a new module, you must specify the module type. Let's recap the 1-letter codes which are used:

- L - LDA
- P - PDA
- G - GDA
- P - program
- N - subprogram
- S - subroutine
- H - helproutine
- C - copycode
- M - map

Consider the following examples:

Example 3: create a new subprogram

```
E N
```

In this example, the Program Editor is invoked since the module to be created is a subprogram.

Example 4: create a new LDA

```
E L
```

In this example, the Data Area Editor is invoked since the module to be created is an LDA. The parameter indicates the module type: L for LDA, A for PDA, G for GDA.

Example 5: create a new map

```
E M
```

In this example, the Map Editor is invoked since the module to be created is a map.

Please be aware that a valid EDIT command with parameters immediately destroys the prior contents of the edit buffer. You are not prompted to save the prior contents.

When the edit command is given without parameters, the current contents of the edit buffer are presented for editing in the appropriate editor. If the edit buffer is empty, the Program Editor is invoked by default. In order to create a new module, the edit buffer must not already contain a module of the type specified. If it does, the contents of the edit buffer will be presented for editing.

NATURAL Program Editor

P

The Program Editor, used to create and maintain programs, subprograms, subroutines, helproutines and copycode is the most frequently used tool in the NATURAL arsenal. The screen is as follows:

```
>                                        > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** SUBPROGRAM IVINSTN1
  0020 ** SUBPROGRAM TO CALCULATE INSTALLMENT AMOUNTS
  0030 **
  0040 ** CREATED ON 01/01/1998 BY SIMPSON
  0050 ************************************************************************
  0060 DEFINE DATA
  0070 PARAMETER USING IVINSTA1 /* PDA
  0080 LOCAL
  0090 1 #FINAL-AMOUNT (P7.2)
  0100 1 #NBR-INSTALLMENTS (P2)
  0110 1 #REG-AMOUNT (P7.2)
  0120 1 #REG-COUNT (P2)
  0130 END-DEFINE
  0140 *
  0150 RESET IVINSTA1.INSTALLMENT-AMOUNT(*)
  0160 *
  0170 #NBR-INSTALLMENTS := IVINSTA1.NBR-INSTALLMENTS
  0180 #REG-AMOUNT := IVINSTA1.TOTAL-PRICE / #NBR-INSTALLMENTS
  0190 #REG-COUNT := #NBR-INSTALLMENTS - 1
  0200 #FINAL-AMOUNT :=
       ....+....1....+....2....+....3....+....4....+....5....+...  S 26    L 1
```

In the top left corner of the screen (between 2 “>“ symbols) is the command field, where commands (except line specific commands) are entered.

Following is a direction indicator with a value of “+”. For certain editor commands, this controls where lines are inserted. A value of “+” (the default) causes lines to be inserted after the line where the command was entered. A value of “-” causes lines to be inserted before.

The type and name of the module being edited, and the name of the library, are shown in the top right corner of the screen.

On the second line is a guide showing column numbers. The caption top preceding the guide indicates that the first line of the module is being displayed. The caption “Bot” would indicate that the last line of the module was being displayed. The caption “All” would indicates that all lines of the module were being displayed.

At the bottom of the screen is another guide showing column numbers.

The information in the bottom right corner indicates the size of the module (in this case, 26 lines) and the relative number of the first line being displayed (in this case, 1).

Program Editor commands

All 3 editors utilize a common command set. For commands available in only one editor (there are few), the command description is placed within that section. For commands available in more than one editor (there are many), the description is placed within this section. For completeness, the description of each editor includes a matrix, summarizing the commands available.

Commands fall into 4 categories:

- editor level commands
- line level commands
- field level commands (Map Editor only)
- terminal commands

Editor level commands generally begin with a letter. They are entered into the editor command field at the top of the screen.

Line level commands generally begin with "." (dot) in the Program Editor and Data Area Editor and ".." (dot dot) in the Map Editor. They are entered on the specific source line where they are to operate. Some commands have both editor level and line level variations.

Field level commands generally begin with "." (dot). They apply only to the Map Editor and are entered on the specific map field where they are to operate. Some Map Editor commands have both line level and field level variations.

Terminal commands may be entered anywhere.

Following is a table detailing the commands available in the Program Editor. Cross-editor availability is indicated in the last 3 columns.

Program Editor commands

Editor Command	Line Command	Description	Editor P	Editor D	Editor M
	.X	MARK AS X Marks a line for later manipulation.	✓	✓	✗
	.Y	MARK AS Y Marks a line for later manipulation.	✓	✓	✗
X		GO TO LINE X Redraws the screen so that the line marked X is visible at or near the top of the screen.	✓	✓	✗
Y		GO TO LINE Y Redraws the screen so that the line marked Y is visible at or near the top of the screen.	✓	✓	✗
	.P	POSITION Redraws the screen so that the line where it is used appears at (or near) the top of the screen.	✓	✓	✗
+ (or PF8)		DOWN Initiates downward scrolling.	✓	✓	✗

		+ Scrolls down 1 screen (about 18 lines). +H Scrolls down half a screen (about 9 lines). +5 Scrolls down 5 lines.			
- (or PF7)		UP Initiates upward scrolling. - Scrolls up 1 screen (about 18 lines). -H Scrolls up half a screen (about 9 lines). -5 Scrolls up 5 lines.	✓	✓	✗
T (or --)		TOP Scrolls to the top (first line) of the module.	✓	✓	✗
B (or ++)		BOTTOM Scrolls to the bottom (last line) of the module.	✓	✓	✗
	.D	DELETE Deletes one or more source lines. .D Deletes the line where the command is used. .D(10) Deletes the line where the command is used plus the next 9 lines.	✓	✓	✗
D		DELETE Deletes marked source lines. DX Deletes the line marked X. DY Deletes the line marked Y. DX-Y Deletes the block of lines bounded by the lines marked X and Y.	✓	✓	✗
E		EXCEPT Deletes lines before the line marked X and/or after the line marked Y. EX	✓	✓	✗

		Deletes all lines before the line marked X. EY Deletes all lines after the line marked Y. EX-Y Deletes all lines **except** the block bounded by the lines marked X and Y.			
	.C	COPY Copies one or more source lines. .C Copies the line where the command is used, inserting the new line immediately below* the line where the command is used. .C(9) Copies the line where the command is used 9 times, inserting the new lines immediately below* the line where the command is used. .CX Copies the line marked X, inserting the new line immediately below* the line where the command is used. .CY(5) Copies the line marked Y 5 times, inserting the new lines immediately below* the line where the command is used. .CX-Y Copies the block of lines bounded by the lines marked X and Y, inserting the new lines immediately below* the line where the command is used. .CX-Y(3) Copies the block of lines bounded by the lines marked X and Y 3 times, inserting the new lines immediately below* the line where the command is used.	✓	✓	✗
	.M	MOVE Moves one or more source lines. .MX Moves the line marked X, inserting it immediately below* the line where the command is used. .MX-Y Moves the block of lines bounded by the lines marked X and Y, inserting them immediately below* the line where the command is used.	✓	✓	✗

	.I	INSERT Inserts one or more lines. .I Inserts several empty lines immediately below* the line where the command is used so that additional lines of source code can be entered. .I(IVEXTRL1) Inserts source lines from module IVEXTRL1 immediately below* the line where the command is used.	✓	✓	✗
	.G	GENERATE Generates source code. Only available if NATURAL CONSTRUCT is installed. *This command is described in more detail below.*	✓	✗	✗
	.L	LET Backs-out any changes made to the line where the command is used, providing the changes were not committed by pressing ENTER.	✓	✓	✗
	.S	SPLIT Splits the line where the command is used. Splits at the cursor position when ENTER is pressed.	✓	✗	✗
	.J	JOIN Joins the line where the command is used and the subsequent line. Joins when ENTER is pressed.	✓	✗	✗
CLEAR		CLEAR Clears the edit buffer, deleting all lines.	✓	✓	✓
READ		REVERT TO SAVED SOURCE Reads the last version of the module saved to disk into the edit buffer, overwriting prior contents.	✓	✓	✓
SET TYPE		SET TYPE Changes the module type. SET TYPE N Changes module type to subprogram.	✓	✓	✗
SC		SCAN/REPLACE Scans for a specified string and optionally replaces it with another. This command is described in more detail below.	✓	✓	✗
C		CHECK Checks for syntax errors. Error messages are displayed with a reference number to facilitate look-up.	✓	✓	✗
?		HELP Provides information. This command is described in more detail below.	✓	✓	✓
STRUCT		STRUCTURE Structures (adjusts indentation of) source code for	✓	✗	✗

		consistency and to increase readability.			
R		RUN Executes the source code in the edit buffer, providing it contains a syntactically correct program module.	✓	✗	✗
SA		SAVE Saves the source code contained in the edit buffer to disk. Does not create an executable module.	✓	✓	✓
CAT		CATALOG Creates an executable module based on the contents of the edit buffer. Does not save the source code.	✓	✓	✓
STOW		STOW Saves the source code contained in the edit buffer to disk and creates a corresponding executable module. Equivalent to SAVE plus CATALOG.	✓	✓	✓
L		LIST Lists source code. L Lists contents of edit buffer. L IVEXTRL1 Lists source code for module IVEXTRL1.	✓	✓	✓
L DIR		LIST DIRECTORY Lists directory information (type size, last update, etc.) about a module. L DIR Lists directory information for the module contained in the edit buffer. L DIR IVEXTRL1 Lists directory information for module IVEXTRL1.	✓	✓	✓
%L terminal command		LOWER CASE When lower case is used, preserves it for certain syntactical elements (comments, literals, etc.).	✓	✓	✗
%U terminal command		UPPER CASE When lower case is used, converts it to upper case.	✓	✓	✗
%% terminal command		EXIT Exits the editor.	✓	✓	✗
.		EXIT Exits the editor.	✓	✓	✓

In the above descriptions, ***below*** * means above or below, depending on the direction indicator.

.G (generate) command

The GENERATE command allows you to quickly generate lines of source code. The command is only available from the Program Editor and only if NATURAL CONSTRUCT is installed. In this course, only two uses will be covered. Consider the following examples:

Example 1: GENERATE view

```
>                                        > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** SUBPROGRAM IVINSTN1
  0020 ** SUBPROGRAM TO CALCULATE INSTALLMENT AMOUNTS
  0030 **
  0040 ** CREATED ON 01/01/1998 BY SIMPSON
  0050 ************************************************************************
  0060 DEFINE DATA
  0070 PARAMETER USING IVINSTA1 /* PDA
  0080 LOCAL
  0090 1 #FINAL-AMOUNT (P7.2)
  0100 1 #NBR-INSTALLMENTS (P2)
  0110 1 #REG-AMOUNT (P7.2)
  0120 1 #REG-COUNT (P2)
       .G(VIEW)
```

In this example, a database file (record) definition is generated. This is similar to using the VIEW command in the Data Area Editor. The name of the view is obtained from a prompt window which opens as follows:

```
            CUVWMA           ***** Generate a View from PREDICT *****         CUVWMA1

              *NATURAL DDM Name..: _____________________________________ (Must be in PREDICT)
              Program View Name..: _____________________________________ (Optional)

              Include C* Variables.................: X
              Include Redefined Components.........: X
              Include Superdescriptor Fields.......: _
              Include NATURAL Format Specifications.: _ (Structure instead of a View)

              ENTR=gen,PF1=help,PF2=retrn,PF3=quit
```

Example 2: GENERATE a CALLNAT statement

```
>                                        > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** SUBPROGRAM IVINSTN1
  0020 ** SUBPROGRAM TO CALCULATE INSTALLMENT AMOUNTS
  0030 **
  0040 ** CREATED ON 01/01/1998 BY SIMPSON
  0050 ************************************************************************
  0060 DEFINE DATA
  0070 PARAMETER USING IVINSTA1 /* PDA
  0080 LOCAL
  0090 1 #FINAL-AMOUNT (P7.2)
  0100 1 #NBR-INSTALLMENTS (P2)
  0110 1 #REG-AMOUNT (P7.2)
  0120 1 #REG-COUNT (P2)
  0130 END-DEFINE
  0140 *
       .G(CALLNAT)
```

In this example, a CALLNAT statement is generated with the right number of parameters in the right sequence. The name of the subprogram is obtained from a prompt window which opens as follows:

```
          CUCNMA    ***** CALLNAT Statement *****   CUCNMA1

            *Subprogram Name: ________              (Required)

            ENTR=gen,PF1=help,PF2=retrn,PF3=quit
```

The generated code is inserted either above or below the line where the command is used, depending on the “direction indicator”, and can be edited as necessary.

SCAN (scan/replace) command

The SCAN/REPLACE command allows you to locate occurrences of a specified string. The Program Editor additionally provides an option to replace all occurrences with a different or null string. Replacing with a null string effectively deletes all occurrences of the string. The direction of the scan (forwards or backwards) is determined by the “direction indicator”.

Consider the following examples:

Example 1: SCAN without parameters

```
> SC                                        > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
```

In this example, a SCAN command without parameters is used from the Program Editor. A window opens as follows:

```
Scan value ...... _____________________________________
Replace value ... _____________________________________

Replace with null value .... _
Absolute scan ............... _
X - Y  range ................ _

                                Press PF3 for Exit
```

Scan value is required. It allows you to specify the string to be found.

Replace value is optional. It allows you to specify the replacement string. If a value is entered, every occurrence of the *Scan value* will be replaced.

Replace with null is optional. By entering X you specify that every occurrence of the *Scan value* is to be replaced with a null string, i.e. deleted.

Absolute scan is optional. By entering X you specify that an absolute scan is required - all occurrences of the *Scan value* (not just those delimited by spaces or certain punctuation symbols) will be found.

X - Y range is optional. By entering X you specify that the scan should be limited to the range of lines bounded by lines marked as X and Y.

Example 2: SCAN with parameters

```
> SC CLIENT-ID                          > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
```

In this example, a SCAN command with a parameter is used from the Program Editor. Since at least one parameter is present, the prompt window does not open.

If a prior SCAN command was issued, the prior values for *Absolute scan* and *X - Y range* will be assumed. If not, default values of FALSE will be assumed.

Null will be assumed for *Replace value* and FALSE will be assumed for *Replace with null value* regardless of the prior SCAN.

Example 3: repeat prior scan

```
> SC =                                  > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
```

In this example, a SCAN command with a parameter is used from the Program Editor. Since at least one parameter is present, the prompt window does not open.

A parameter value of “=“ is only valid when a prior SCAN command was issued. The prior values for *Scan value*, *Absolute scan* and *X - Y range* will be assumed.

Null will be assumed for *Replace value* and FALSE will be assumed for *Replace with null value* regardless of the prior SCAN.

? (help) command

The HELP command allows you to obtain information. It is commonly used to obtain information about an error message for which you already have a reference number. Consider the following example:

```
> ? 39                                    > +  Subprogram  IVINSTN1 Lib INVLIB
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
```

In this example, information about “error 39” would be returned, as follows:

```
           00:00:01             *** NATURAL HELP FACILITY ***                 98-01-01
                                 - Error Message Nr. NAT0039 -                 Page 1

              Sum of field lengths in REDEFINE > length of base field.

           Tx *** Short Text ***

              Sum of field lengths in REDEFINE > length of base field.

           Ex *** Explanation ***

              The accumulated length of all new fields in a REDEFINE statement
              must be less than or equal to the length of the base field, counted in
              bytes.









           + <--- Enter code (M Q T P + - .)          More ...
           PF1=Help,PF2=Menu,PF3=Exit,PF5=Print,PF12=Canc
```

NATURAL Data Area Editor

D

The Data Area Editor is used to create and maintain LDAs, PDAs and GDAs. The screen is as follows:

```
Parameter IVINSTA1  Library INVLIB                          DBID   1 FNR   1
Command                                                              > +
I T L Name                            F Leng Index/Init/EM/Name/Comment
All - ---------------------------------- - ---- ---------------------------------
    1 IVINSTA1
    2 INPUT-PARAMETERS
    3 TOTAL-PRICE                     P  7.2
    3 NBR-INSTALLMENTS                P    2
    2 OUTPUT-PARAMETERS
    3 INSTALLMENT-AMOUNT              P  7.2 (1:12)









------------------------------------------------------------------- S 6    L 1
```

The type and name of the module being edited, and the name of the library, are shown in the top left corner of the screen.

The information in the top right corner indicates the database and file where the module is stored.

On the second line is the command field, where commands (except line specific commands) are entered.

Following is a direction indicator with a value of “+”. For certain editor commands, this controls where lines are inserted. A value of “+” (the default) causes lines to be inserted after the line where the command was entered. A value of “-” causes lines to be inserted before.

The third line contains column headings. “I” stands for information (supplied by the editor and not changeable. “T” stands for type. “L” stands for level. “F” stands for format. “Leng” stands for length. “EM” stands for edit mask.

Within the Data Area Editor, the last column has multiple uses as suggested by the column heading. Essential uses are illustrated by examples throughout this course.

The information in the bottom right corner indicates the size of the module (in this case, 6 lines) and the relative number of the first line being displayed (in this case, 1).

Data Area Editor commands

This section covers the essential Data Area Editor commands.

Since all 3 editors utilize a common command set, many of the available commands have already been described under the heading Program Editor Commands. To avoid repetition, such commands will only be summarized here. Please refer back to the full descriptions as necessary.

Data Area Editor commands					
Editor Command	Line Command	Description	Editor		
			P	D	M
	.X	MARK AS X	✓	✓	✗
	.Y	MARK AS Y	✓	✓	✗
X		GO TO LINE X	✓	✓	✗
Y		GO TO LINE Y	✓	✓	✗
	.P	POSITION	✓	✓	✗
+ (or PF8)		DOWN	✓	✓	✗
- (or PF7)		UP	✓	✓	✗
T (or --)		TOP	✓	✓	✗
B (or ++)		BOTTOM	✓	✓	✗
	.D	DELETE	✓	✓	✗
D		DELETE	✓	✓	✗
E		EXCEPT	✓	✓	✗
	.C	COPY	✓	✓	✗
	.M	MOVE	✓	✓	✗
	.I	INSERT	✓	✓	✗
	.L	LET	✓	✓	✗
CLEAR		CLEAR	✓	✓	✓
READ		REVERT TO SAVED SOURCE	✓	✓	✓
SET TYPE		SET TYPE	✓	✓	✗
SC		SCAN/REPLACE	✓	✓	✗
C		CHECK	✓	✓	✗
?		HELP	✓	✓	✓
SA		SAVE	✓	✓	✓
CAT		CATALOG	✓	✓	✓
STOW		STOW	✓	✓	✓
L		LIST	✓	✓	✓
L DIR		LIST DIRECTORY	✓	✓	✓
%L		LOWER CASE	✓	✓	✗
%U		UPPER CASE	✓	✓	✗
%%		EXIT	✓	✓	✗
.		EXIT	✓	✓	✓
	.V	VIEW Incorporates selected fields from a database record definition into a data area. *This command is described in more detail below.*	✗	✓	✗

.V (view) command

The VIEW command enables you to incorporate selected fields from a database record definition into a data area. Consider the following example:

```
.V(CLIENT)
```

In this example, a view of database file CLIENT is brought in. You are prompted to mark (with X) the specific fields to be incorporated. The screen is as follows:

```
SYSGDA 4001: Mark fields to incorporate into data area.
Local              Library INVLIB                          DBID   1 FNR   1
View CLIENT
I T L Name                              F Leng Index/Init/EM/Name/Comment
------------------------------------------------------------------------------
X   2 CLIENT-ID                        N    8
X   2 BILLING-NAME                     A   40
    2 BILLING-ADDRESS                  A   40 (1:3)
    2 AUDIT-TIME                       T
    2 AUDIT-USER                       A    8
    2 AUDIT-PROGRAM                    A    8
    2 AUDIT-COUNTER                    P    7
```

The incorporated lines are inserted either above or below the line where the command is used, depending on the “direction indicator”.

The VIEW command is the only way to incorporate a database file (record) definition into an LDA, PDA or GDA. The incorporated definition cannot be edited at the field level. If required, the entire view can be deleted using a DELETE command on the first line of the view.

NATURAL Map Editor

M

When the Map Editor is invoked (e.g. E M from the NATURAL command line), the main menu is presented:

```
00:00:01                ***** NATURAL MAP EDITOR *****                01/01/98
User XXX999                      - Edit Map -                  Library INVLIB

                   Code    Function
                   ----    ---------------------------------
                     D     Field and Variable Definitions
                     E     Edit Map
                     I     Initialize new Map
                     H     Initialize a new Help Map
                     M     Maintenance of Profiles & Devices
                     S     Save Map
                     T     Test Map
                     W     Stow Map
                     ?     Help
                     .     Exit



              Code .. I     Name .. ________       Profile .. SYSPROF_

Command ===>
PF1=Help,PF3=Exit,PF4=Test,PF5=Edit
```

We'll review the essential options on this menu later. For now, we'll focus on creating a new map module. The default code of I initializes a new map module and then presents it for editing. You must specify a profile or accept the default profile of SYSPROF.

Map profiles

When creating a new map module, a profile is used to control the default map settings for such things as:

- number of lines and columns
- identifying different classes of fields
- assigning intensity
- upper/lower case
- filler character for input/modifiable fields

Profiles help to provide consistent map settings and, by extension, consistent programmer and user interfaces. Any of the default map settings can be overridden.

You can edit the profile or change current map settings at any time, by entering a code of M (maintenance of profiles & devices) on the above menu. The returned screen looks something like this:

```
00:00:01                    Map Profile Maintenance                    01/01/98

                Code   Function
                ----   ----------------------------------
                  A    Add Profile
                  M    Modify Profile
                  D    Display Profile
                  P    Purge Profile

                  AD   Add Device
                  MD   Modify Device
                  DD   Display Device
                  PD   Purge Device

                  C    Change Map Settings of Current Map

                  . ?  Exit, Help

          Code .. __           Profile .. SYSPROF_      Device .. ________


PF3=Exit
```

This screen allows you to create and maintain profiles. It also allows you to change the current map settings. When you enter a code of C (change map settings of current map), a screen similar to the following is returned:

```
00:00:01              Define Map Settings for MAP                    01/01/98

  Delimiters          Format                          Context
  -----------------   -----------------------------   -------------------------
  Cls Att CD  Del     Page Size ...... 23             Device Check .... ________
   T   D      BLANK   Line Size ...... 79             WRITE Statement   _
   T   I      ?       Column Shift ... 0 (0/1)        INPUT Statement   X
   A   D      _       Layout ......... ________       Help  ___________________
   A   I      )        dynamic ....... N (Y/N)         as field default N (Y/N)
   A   N      ¬       Zero Print ..... N (Y/N)
   M   D      &       Case Default ... UC (UC/LC)
   M   I      :       Manual Skip .... N (Y/N)        Automatic Rule Rank  1
   O   D      +       Decimal Char ... .              Profile Name .... SYSPROF
   O   I      (       Standard Keys .. N (Y/N)
                      Justification .. L (L/R)        Filler Characters
                      Print Mode ..... __             ------------------------
                                                      Optional, Partial ....
                      Control Var .... ________       Required, Partial ....
                                                      Optional, Complete ...
                                                      Required, Complete ...

 PF1=Help,PF3=Exit,PF12=Let
```

The above screen is presented automatically when you initialize a new map allowing you to override any of the default map settings.

The essential map settings are:

- Delimiters (explained below)
- Page Size (number of rows)
- Line Size (number of columns)
- Case Default (whether input data is converted to upper case)
- Control Var (name of map level control variable)
- WRITE Statement (whether map is used in conjunction with WRITE, as opposed to INPUT)
- INPUT Statement (whether map is used in conjunction with INPUT, as opposed to WRITE)
- Filler Characters (to indicate location and length of input/modifiable fields)

Map fields

A map consists of a series of fields. Usually this is a mixture of text fields (e.g. headings and prompts), output fields (displayed to the user, but not changeable), modifiable fields (changeable) and input fields. Non-text fields are attached to data items within the “calling” module.

Map field delimiters

The delimiter characters used to signal the start of a field, its class and its intensity, are determined by the current map settings. The edit screen shows the available delimiters in the top right corner:

```
Ob _                                      Ob D CLS ATT  DEL    CLS ATT  DEL
.                                         .     T  D    Blnk    T  I    ?
.                                         .     A  D    _       A  I    )
.                                         .     A  N    ¬       M  D    &
.                                         .     M  I    :       O  D    +
.                                         .     O  I    (
.                                         .
001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----


Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

There is no explicit delimiter to signal the end of a field. Implicitly, a null/space character or another delimiter signals the end of a field.

In this case, the delimiters of interest are:

- blank to identify a text field with default intensity
- “?” to identify a text field with high intensity
- “¬” to identify a non-display input field
- “&” to identify a modifiable field with default intensity
- “:” to identify a modifiable field with high intensity
- “+” to identify an output field with default intensity
- “(“ to identify an output field with high intensity

As an alternative to the default “half screen” mode shown above, you can press PF9 to switch to “full screen” mode. PF9 toggles between the two modes. In “full screen” mode the delimiter settings are not displayed.

Creating a map module

This section explains the steps for creating a map module.

Step 1 - Initialize new map

Select option I, enter a module name (mandatory) and a profile name (optional). Press ENTER.

Step 2 - Adjust map settings

Adjust map settings, overriding the defaults obtained from the profile, as necessary. Press ENTER.

Step 3- Create text fields

Since text fields are easy to move around, you may find it helpful to initially create all map fields as text fields. When you're happy with the layout, you can then replace some of the text fields with non-text fields, as appropriate.

This is best done using "full screen" mode. PF9 toggles between "half screen" and "full screen" modes.

Consider the following example:

```
                          ?Client?Information?Screen


                      Action: X (A, C, D, M, N, P)

                   Client ID: 99999999

                Billing Name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

              Billing Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


                Last Updated: XXXXXXXXXXXXXXXXXX

                        User: XXXXXXXX

                     Program: XXXXXXXX


001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----
```

This map contains only text fields. You can type them in exactly as you want them to appear and/or move them around using some of the commands detailed below.

Notice that the heading “Client Information Screen” has a ? delimiter at the beginning of each word. Effectively, each word is treated as a different text field and all 3 are intensified.

Notice also that a colon can be used as regular text - it is not always treated as a delimiter.

A good technique when designing screens is to center the longest line, then align map fields on other lines accordingly. The above screen layout illustrate the results. Creating all map fields initially as text fields facilitates use of this technique.

Step 4 - Save your work (optional)

Press PF3 to return to the Map Editor menu and save your work. I recommend doing this frequently during an edit session.

Step 5 - Create non-text fields and assign to data items

If necessary, toggle to “half screen” mode using PF9 so that the screen appears as follows:

```
Ob _                                    Ob D CLS ATT  DEL    CLS ATT  DEL
.                                       .     T  D    Blnk    T  I    ?
.                                       .     A  D    _       A  I    )
.                                       .     A  N    ¬       M  D    &
.                                       .     M  I    :       O  D    +
.                                       .     O  I    (
.                                       .
001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

                           ?Client?Information?Screen


                      Action: X (A, C, D, M, N, P)

                   Client ID: 99999999

                Billing Name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

             Billing Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

The Ob (object) prompt in the top left corner allows you to list the relevant parts of another source module and reference the data fields or variables directly.

Type in the module type (P for program, N for subprogram or L for LDA, etc.) followed by the module name and press ENTER. This lists the relevant parts of the specified module.

```
Ob P IVCLNTP1                              Ob D CLS ATT  DEL    CLS ATT  DEL
1 #CV-CLIENT-ID                            .     T  D    Blnk   T  I    ?
2 #ISN                                     .     A  D    _      A  I    )
. #MAP                                     .     A  N    ¬      M  D    &
3 #ACTION                                  .     M  I    :      O  D    +
4 CLIENT-ID                                .     O  I    (
5 BILLING-NAME                             .
001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

                          ?Client?Information?Screen



                     Action:&3 (A, C, D, M, N, P)

                  Client ID:&4

               Billing Name:&5

            Billing Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

For each map field which relates to a data item, do the following:

1. position the cursor at the character immediately preceding the first character of the map field
2. select and type the appropriate delimiter (e.g. &)
3. type the relevant data item number (or letter) from the top portion of the screen (e.g. 3)
4. space over the text representing the field (e.g. XXX)

Press ENTER to attach the data items to the map fields.

To scroll through the map fields, press PF8 or PF7 until the required map fields are displayed.

To scroll through the list of data items, type + or - following the Ob (object) prompt until the required data items are displayed.

By positioning the cursor on the name of the module and pressing PF1, you can invoke a little-publicized Map Editor feature giving you the ability to:

- view a full screen list of available data items
- scroll through the full screen list a page at a time
- select a field from the long list as the top of the normal list

Notice that there's a second Ob (object) prompt in the top right corner of the screen, where delimiter information is displayed by default. This allows you to reference a second source module simultaneously.

Step 6 - Save your work (optional)

As per step 2.

Step 7 - Fine-tune using Map Editor commands

Use the Map Editor commands described in the next section to make any necessary adjustments to the map.

Map Editor commands

This section covers the essential Map Editor commands.

Since all 3 editors utilize a common command set, many of the available commands have already been described under the heading Program Editor Commands. To avoid repetition, such commands will only be summarized here. Please refer back to the full descriptions as necessary.

The table below starts with 3 columns headed *Editor Command*, *Line Command* and *Field Command*. This should be interpreted as follows:

- Commands listed in the *Editor Command* column are entered directly into the NATURAL command line.
- Commands listed in the *Line Command* column are entered into the first few columns of the specific line where they are to operate, overtyping existing map fields if necessary. If appropriate, the map fields will be restored automatically after the command is executed.
- Commands listed in the *Field Command* column are entered at the specific field where they are to operate, starting at the character position occupied by the delimiter. If appropriate, the field will be restored automatically after the command is executed.

Map Editor commands

Editor Command	Line Command	Field Command	Description	Editor P	Editor D	Editor M
CLEAR			CLEAR	✓	✓	✓
READ			REVERT TO SAVED SOURCE	✓	✓	✓
?			HELP	✓	✓	✓
SA			SAVE	✓	✓	✓
CAT			CATALOG	✓	✓	✓
STOW			STOW	✓	✓	✓
L			LIST	✓	✓	✓
L DIR			LIST DIRECTORY	✓	✓	✓
.			EXIT	✓	✓	✓
	..S		SPLIT Splits the line where the command is used. Splits at the cursor position when ENTER is pressed.	✗	✗	✓

	..J		JOIN Joins the line where the command is used and the subsequent line. Joins when ENTER is pressed.	✗	✗	✓
	..C		CENTER Centers the line where the command is used.	✗	✗	✓
	..I		INSERT Inserts one or more empty lines. ..I Inserts an empty line immediately below* the line where the command is used. ..I(2) Inserts 2 empty lines immediately below* the line where the command is used.	✗	✗	✓
	..D		DELETE Deletes one or more lines. ..D Deletes the line where the command is used. ..D(3) Deletes the line where the command is used plus the next 2 lines.	✗	✗	✓
		.D	DELETE Deletes a field.	✗	✗	✓
		.D .D	DELETE (multiple fields) When 2 commands are entered in the same transaction, deletes a block of fields.	✗	✗	✓
		.M	MOVE Moves a field. The new position of the field is indicated by the cursor position when ENTER is pressed.	✗	✗	✓
		.M .M	MOVE (multiple fields) When 2 commands are entered in the same transaction, moves a block of fields. The new position of the first is indicated by the cursor position when ENTER is pressed.	✗	✗	✓
		.E	EDIT FIELD DEFINITION Allows editing of a field definition. *This command is described in more detail below.*	✗	✗	✓
		.A	EDIT ARRAY DEFINITION Allows editing of an array definition. *This command is described in more detail below.*	✗	✗	✓

.E (edit field definition) command

The .E (edit field definition) command allows you to change the definition of a field. In response to this command, field information is displayed in the top part of the edit screen. Consider the following example:

```
Fld #MAP.#ACTION                                                    Fmt A1
 ------------------------------------------------------------------------------
 AD= MDT'_'____     ZP=         SG=        HE= _____________________   Rls 0
 AL= _____          CD= __      CV= _____________________________   Mod Data
 PM= __             BX= ____    DY= _____________________________
 EM= ____________________________________________________________

001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

                          ?Client?Information?Screen


                     Action:.E (A, C, D, M, N, P)

                  Client ID: 99999999

               Billing Name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

            Billing Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

In this example, map field ACTION is edited.

The .E (edit field definition) command allows any of the following to be changed:

- data item attachment
- format and length
- attribute definition (AD)
- zero print option (ZP)
- sign position (SG)
- helproutine (HE)
- alphanumeric length(AL)
- color definition (CD)
- control variable
- dynamic attributes
- edit mask

PM (print mode), BX (box definition) and DY (dynamic attributes) are advanced features outside the scope of this course.

Map field attribute definition

The AD parameter allows you to change the attribute definition of the map field. Each attribute is represented by a single letter. The essential values are as follows:

Attribute definition		
Value	**Description**	**Attribute Type**
A	input	input/output
M	modifiable	
O	output	
P	modifiable, temporarily protected	
B	blinking	field representation
C	cursive/italic	
D	default intensity	
I	intensified	
N	non-display	
U	underlined	
V	reverse video	
L	left justified	field alignment
R	right justified	
Z	right justified with leading zeros	
T	translate to upper case	upper/lower case
W	preserve lower case	
‘_’	filler character for input/modifiable fields	filler character

Attributes of the same type are mutually exclusive.

Map field zero print option

The ZP parameter allows you to control whether a zero value is shown or suppressed. The possible values are as follows:

Zero print option	
Value	**Description**
ON	a zero value is shown
OFF	a zero value is suppressed

Map field sign position

The SG parameter allows you to specify whether a character position should be reserved for a sign. Applies only to numeric fields. The possible values are as follows:

Sign position	
Value	**Description**
null	a character position will be reserved
OFF	a character position will not be reserved - a negative value will be shown as a positive value

Map field helproutine

The HE parameter allows you to specify the name of a field-level helproutine. This could be a map module or an executable helproutine module. The specified helproutine will be executed if the user enters a help character (“?” by default) in the first character position of the field or positions the cursor anywhere on the field and presses the help key (PF1 by convention).

The name of the helproutine module must be specified in quotes. Optionally, parameter data, which is passed to the helproutine module, may also be specified. There is a limit of 20 such parameters, and each must be separated by a comma. Parameters may be literal values enclosed in quotes or the names of map fields.

It’s important to realize that helproutines are executed before other map data is processed. This means that helproutines cannot access map data which was entered by the user in the same transaction.

A parameter consisting only of an equals sign (not enclosed in quotes), causes the map field name to be passed (in A65 format) to the helproutine module. This is useful for sites which have a “Help Central” module through which all help requests are channeled.

In addition to the explicit parameters, an implicit parameter is also passed to the helproutine module, representing the map field for which help is being generated. “Active help” modules, which display a list of possible values and allow the user to select one, return the user-selected value via the implicit parameter causing the map field for which help was invoked to be populated.

If the HE parameter is null and the user invokes field-level help, screen-level help may be given instead. Screen-level help is defined in the same way, but using the map profile screen.

Map field alphanumeric length

The AL parameter allows you to define a map field which is shorter than the attached data item. The rightmost characters will be truncated.

Map field color definition

The CD parameter allows you to specify a color, assuming appropriate hardware is available. The possible values are as follows:

Color definition	
Value	**Description**
BL	blue
GR	green
NE	neutral (white)
PI	pink (magenta)
RE	red
TU	turquoise (cyan)
YE	yellow

Map field edit mask

The EM parameter allows you to specify an edit mask which will be used to format output and validate input. Since the Map Editor is case-sensitive, be sure to explicitly enter characters in upper case, where appropriate.

The default value for each of the above parameters varies from site to site.

When editing is complete, press ENTER to return to the edit screen.

When editing field definitions, PF5 can be used to select the next non-text field and PF4 can be used to select the prior field.

.A (edit array definition) command

The .A (edit array definition) command allows you to define or change the definition of an array of map fields.

The repeating map fields are attached to a repeating data item, allowing multiple occurrences of the data to be displayed. You can specify which occurrences to display and in what format. Consider the following examples:

Example 1: 1-dimensional array

To create an array of map fields for data item BILLING-ADDRESS, which is defined as (A40/3), we would enter a .A command against the field. In response, the following screen would be returned:

```
Name #MAP.BILLING-ADDRESS                        Upper Bnds 3_____ 1_____ 1_____
-------------------------------------------------------------------------------
Dimensions                        Occurrences    Starting from     Spacing
1 . Index vertical                1__            ________          0   Lines
0 . Index horizontal              1__            ________          1   Columns
0 . Index (h/v) V                 1__            ________          0   Cls/Ls

001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

                          ?Client?Information?Screen


                     Action: X (A, C, D, M, N, P)

                  Client ID: 99999999

               Billing Name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

             Billing Address:.AXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

The “upper bounds” information in the top right corner of the screen shows the dimensions of the array - in this case, a 1-dimensional array with 3 occurrences.

It makes sense to arrange the map fields vertically. Since this is the default, we would only need to specify the number of occurrences to be displayed, the index value for the first occurrence to be displayed and the number of blank lines between each occurrence. This would be done as follows:

```
Name #MAP.BILLING-ADDRESS                        Upper Bnds 3_____ 1_____ 1_____
-------------------------------------------------------------------------------
Dimensions                        Occurrences    Starting from     Spacing
1 . Index vertical                3__            1_______          0   Lines
0 . Index horizontal              1__            ________          1   Columns
0 . Index (h/v) V                 1__            ________          0   Cls/Ls

001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

                          ?Client?Information?Screen



                     Action: X (A, C, D, M, N, P)

                  Client ID: 99999999

               Billing Name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

             Billing Address:.AXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

After pressing ENTER, the edit screen would show the additional map fields:

```
                          ?Client?Information?Screen


                     Action: X (A, C, D, M, N, P)

                  Client ID: 99999999

               Billing Name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

            Billing Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                             XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                             XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

               Last Updated: XXXXXXXXXXXXXXXXXX

                       User: XXXXXXXX

                    Program: XXXXXXXX


001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----
```

Example 2: 2-dimensional array

To create an array of map fields for data item #DOLLARS, which is defined as (P7.2/1:5,1:12), we would enter a .A command against the field. In response, the following screen would be returned:

```
Name #DOLLARS                                  Upper Bnds 5_____ 12____ 1_____
--------------------------------------------------------------------------------
Dimensions                      Occurrences    Starting from     Spacing
1 . Index vertical              1__            ________          0   Lines
0 . Index horizontal            1__            ________          1   Columns
0 . Index (h/v) V               1__            ________          0   Cls/Ls

001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

 Dollars:.A999999.99




 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

The “upper bounds” information in the top right corner of the screen shows the dimensions of the array - in this case, a 2-dimensional array with 5 x 12 occurrences (5 departments x 12 months).

It makes sense to arrange the departments horizontally and the months vertically. This would be done as follows:

```
Name #DOLLARS                                   Upper Bnds 5_____  12____  1_____
-------------------------------------------------------------------------------
Dimensions                       Occurrences    Starting from     Spacing
2 . Index vertical               12_            1_______          0   Lines
1 . Index horizontal             5__            1_______          3   Columns
0 . Index (h/v) V                1__            ________          0   Cls/Ls

001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----

 Dollars:.A999999.99











Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help  Mset  Exit  Test  Edit  --    -     +     Full  <     >     Let
```

The “1” entered prior to the words “Index horizontal” specifies the first dimension (department, 5 occurrences) is horizontal.

The “2” entered prior to the words “Index vertical” specifies the second dimension (month, 12 occurrences) is vertical.

The “3” entered prior to the word “Columns” specifies 3 spaces between each column.

After pressing ENTER, the edit screen would show the additional map fields:

```
Dollars:+9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99
        +9999999.99  +9999999.99  +9999999.99  +9999999.99  +9999999.99

001   --010---+----+----+---030---+----+----+---050---+----+----+---070---+----
```

Map Editor main menu

Finally, let's revisit the Map Editor main menu to explore the other options:

```
00:00:01                ***** NATURAL MAP EDITOR *****                01/01/98
User XXX999                      - Edit Map -                  Library INVLIB

                  Code    Function
                  ----    --------------------------------
                    D     Field and Variable Definitions
                    E     Edit Map
                    I     Initialize new Map
                    H     Initialize a new Help Map
                    M     Maintenance of Profiles & Devices
                    S     Save Map
                    T     Test Map
                    W     Stow Map
                    ?     Help
                    .     Exit



            Code .. _     Name .. IVCLNTM1       Profile .. SYSPROF_

Command ===>
PF1=Help,PF3=Exit,PF4=Test,PF5=Edit
```

When editing, PF3 returns you to this menu.

D - Field and Variable Definitions

This option allows you to review non-text map fields. A summary screen is returned as follows:

```
00:00:01          Field and Variable Definitions - Summary           01/01/98

 Cmd Field Name (Truncated)                              Mod Format Ar Ru Lin Col
 ___ #MAP.#ACTION_______________________________________ D   A1             6  29
 ___ #MAP.CLIENT-ID_____________________________________ D   N8             8  29
 ___ #MAP.BILLING-NAME__________________________________ D   A40           10  29
 ___ #MAP.BILLING-ADDRESS_______________________________ D   A40    A      12  29
 ___ #MAP.#AUDIT-TIME___________________________________ U   A19           16  29
 ___ #MAP.AUDIT-USER____________________________________ D   A8            18  29
 ___ #MAP.AUDIT-PROGRAM_________________________________ D   A8            20  29
```

In some scenarios (related to helproutines and helpscreens), the Map Editor will add fields to the map module which are not displayed on this screen. Data for these fields is required from the “calling” module. Normally, you only become aware of these if the “calling” module won’t stow.

To see a list of such fields, press PF9 from this screen. If the help feature was invoked unintentionally, you can enter D in the Cmd (command) column to delete the offending fields from the map.

E - Edit Map

This option allows you to edit an existing map.

I - Initialize new Map

This option allows you to initialize a new map from an existing profile.

H - Initialize a new Help Map

This option allows you to initialize a new help map from an existing profile. The default profile for a help map may be different from the default profile for a regular map.

Use of this option identifies the map as a help map and causes some editing restrictions to be invoked.

M - Maintenance of Profiles & Devices

This option allows you to create a new profile or change an existing one.

S - Save Map

This option allows you to save the source version of the map.

T - Test Map

This option allows you to review how the map will look when presented to the user.

W - Stow Map

This option allows you to create an object version of the map.

? - Help

This option allows you to access Map Editor help information.

. - Exit

This option allow you to exit the Map Editor.

The NATURAL command line appears near the bottom of the screen.

The following direct commands are especially useful when working with the Map Editor: READ to revert to a saved version of the map, L to list other modules, EXECUTE to execute a program which uses the map and E to start editing a module other than a map.

(page intentionally blank)

Appendix B
Naming Guidelines

Appendix B – Naming Guidelines

Using consistent and meaningful names greatly enhances the readability and maintainability of NATURAL code. The following guidelines (or similar ones) have worked extremely well in many of the sites with which I've been involved.

General

- use descriptive but succinct names (e.g. CLIENT.BILLING-NAME)
- use hyphens to separate name elements (e.g. BILLING-NAME, #NBR-INSTALLMENTS, CALC-AMOUNT-DUE)
- use consistent abbreviations (e.g. #NBR-INSTALLMENTS, #NBR-DAYS)
- for maximum readability (and minimum ambiguity), qualify datanames (e.g. CLIENT.BILLING-NAME)

Database files/fields

- for filenames, use the singular form to increase the readability of your code (e.g. CLIENT, PRODUCT)
- avoid symbols other than hyphen
- use consistent field names across files (e.g. CLIENT.CLIENT-ID, RECEIVABLE.CLIENT-ID)

Sequential files/fields

- as per database files/fields

Work variables

- give work variables which hold data from a file the same name as the field on the file, prefixed with # (e.g. INVOICE.DUE-AMOUNT and #DUE-AMOUNT)
- avoid using datanames which incorrectly imply a relationship to a field on a file
- in general, prefix all datanames with # (e.g. #MSG)
- to facilitate MOVE BY NAME from a file record to work variables, drop the # prefix, but make the variables part of a group which does have a name starting with # (e.g. CLIENT.CLIENT-ID and #MAP.CLIENT-ID)
- in a group which has a mixture of database-related variables and other variables, retain the # prefix on the latter (e.g. #MAP.CLIENT-ID and #MAP.#ACTION)
- avoid symbols other than hyphen and pound

Global variables

- as per work variables, but substitute ## for #

Subroutines

- for readability, use names that start with a verb (e.g. CALC-AMOUNTS-DUE, PULL-CLIENT-INFO)
- external subroutines names must be unique within the library
- in-line subroutine names must be unique within the module
- avoid symbols other than hyphen

Modules

- module names must be unique within the library
- avoid symbols
- follow the naming standards for your site
- names uniquely identify each module and generally also indicate the module type and associated application
- for this course, module names are in the format AAXXXXTn, where:
  - AA represents 2 alpha characters indicating the application (e.g. IVPRNTP1 - invoicing)
  - XXXX represents a name to describe the function of the module or a number to merely identify it (e.g. IVEXTRP1 - extract, IVPRNTP1 - print, IVCLNTP1 - client)
  - T indicates the module type (P for program, N for subprogram, etc. per Software AG codes) (e.g. IVPRNTP1 - program, IVPRNTM3 - map)
  - n indicates the sequence number (1 for first/only, 2 for second, 3 for third, etc.) (e.g. IVPRNTM1 - first map, IVPRNTM2 - second map)
- related modules should have names which reflect that relationship (e.g. IVPRNTP1 and IVPRNTM1, IVINSTN1 and IVINSTA1)

PDAs

- for readability, start PDAs with a group name equal to the module name (e.g. IVINSTA1)

(page intentionally blank)

Appendix C
NATURAL Quiz

Appendix C – NATURAL Quiz

This quiz is designed to test your knowledge of NATURAL. The questions are typical of those asked in technical interviews by prospective employers.

Following each question are references to the relevant sections of the course. Explicit answers are not given, however you can submit your own answers on-line at www.spsimpson.com/natural/quiz. Time permitting, I'll review your answers and let you know which topics need additional study. This free service is available only to students who have submitted a Course Evaluation. A Course Evaluation form is located at the end of this document.

1. What's an ISN?
- Part 1, ISN
- Part 3, Program IVCLNTP1

2. What's a PE and why should you use it?
- Part 1, Repeating fields

3. What's an MU? How is it different from a PE?
- Part 1, Repeating fields
- Part 3, File specifications

4. How would you determine the number of occurrences that are populated in an MU?
- Part 1, Repeating fields
- Part 1, Counters
- Part 3, Program IVEXTRP1

5. If you are writing an update program to read a product file and increase the price of every product by 5%, and product code is a descriptor, how would you write the program so that it is most efficient?
- Part 2, Database access statements

6. What is a GDA and why would you use one?
- Part 1, GDAs

7. When would you use a subprogram?
- Part 1, Subprograms
- Part 3, Program IVINSTN1
- Part 3, Program IVEXTRP1

8. In what order are records presented when using a FIND statement?
- Part 2, FIND statement

9. In terms of record presentation, what's the difference between READ and FIND?

Part 2, READ statement
Part 2, FIND statement

10. If a superdescriptor includes a field that is null suppressed, what is the effect on the superdescriptor if that field contains a zero or blank value?

Part 1, Keys

11. What's the difference between DISPLAY and WRITE?

Part 2, DISPLAY statement
Part 2, WRITE statement

12. What's a HISTOGRAM used for?

Part 2, HISTOGRAM statement

13. What is .S used for?

Appendix A, NATURAL Program Editor

14. What is .J used for?

Appendix A, NATURAL Program Editor

15. What is transaction data (sometimes called "ET data")? How is it used?

Part 2, END TRANSACTION statement
Part 2, GET TRANSACTION DATA statement
Part 3, Program IVEXTRP1

16. What is CALLNAT used for?

Part 2, CALLNAT statement
Part 3, Program IVEXTRP1

17. What is a FETCH RETURN used for? What's the difference between FETCH and FETCH RETURN?

Part 2, FETCH statement
Part 2, FETCH RETURN statement

18. What is .G used for?

Appendix A, NATURAL Program Editor

19. If you have to update a field that's not a descriptor, and only 10% of the records need to be updated , how would you go about updating to avoid putting all the records in the file on hold?

Part 2, Database access statements

20. Why would you use an in-line subroutine?

Part 1, Subroutines
Part 1, In-line subroutines
Part 3, Program IVEXTRP1

21. Why would you use an external subroutine?

Part 1, Subroutines
Part 1, External subroutines

22. What are the pros and cons of using copycode?

Part 1, Copycode

Course Evaluation

Congratulations! you've completed the course. Won't you now please take a few minutes to give some feedback. Your input will make this course better.

Course Evaluation

Please complete an evaluation form on-line at www.spsimpson.com/feedback or E-mail your responses to training@spsimpson.com. Alternatively, complete and fax both pages to 815 461-3522. Thank you.

1. Your background
- ❍ novice programmer
- ❍ experienced programmer
- ❍ training coordinator / trainer
- ❍ none of the above

2. How did you discover the course?
- ❍ search engine
- ❍ ADABAS/NATURAL Webring
- ❍ link from another site
- ❍ SAG-L list
- ❍ installed on LAN
- ❍ referred by employer (not installed on LAN)
- ❍ referred by a co-worker
- ❍ referred by author
- ❍ none of the above

3. Why did you take the course?
- ❍ new job/opportunity
- ❍ as a refresher
- ❍ employer required it
- ❍ none of the above

4. In what country did you take the course?

5. How long did you spend studying the course?
- ❍ less than 1 hour
- ❍ 1 to 8 hours
- ❍ 8 to 16 hours
- ❍ 16 to 24 hours
- ❍ 24 to 36 hours
- ❍ 36 to 48 hours
- ❍ more than 48 hours

6. Form of study
- ❍ website (www.spsimpson.com)
- ❍ using local area network (LAN)
- ❍ using Acrobat Reader
- ❍ using printed hardcopy
- ❍ formal instructor-led training

7. Course design (look & feel, organization of material, etc.)
- ❍ excellent
- ❍ good
- ❍ satisfactory
- ❍ poor
- ❍ abysmal

8. Course content (range of topics, depth of information etc.)
- ❍ excellent
- ❍ good
- ❍ satisfactory
- ❍ poor
- ❍ abysmal

9. Course quality (writing, examples, etc.)
- ❍ excellent
- ❍ good
- ❍ satisfactory
- ❍ poor
- ❍ abysmal

10. Ease of use
- ❍ excellent
- ❍ good
- ❍ satisfactory
- ❍ poor
- ❍ abysmal

Please check one box only in response to each question.

11. Usefulness to you

- ❍ very useful
- ❍ useful
- ❍ somewhat useful
- ❍ useless
- ❍ totally useless
- ❍ don't know

12. Did/will you download the course to your PC?

- ❍ yes
- ❍ no
- ❍ what is that?

13. Which topics, if any, should be added or expanded?

continue on a separate sheet

14. Which topics, if any, should be deleted or condensed?

continue on a separate sheet

15. Best thing about the course?

continue on a separate sheet

16. Worst thing about the course?

continue on a separate sheet

17. Overall assessment

- ❍ excellent
- ❍ good
- ❍ satisfactory
- ❍ poor
- ❍ abysmal

18. General comments

continue on a separate sheet

19. Personal information (optional, confidential)

Name:

Organization:

Address:

Phone/Fax/E-mail:

Thanks for your input.