TIFORM Reference Manual

Part No. 2234391-9701 **
February 1984

TEXAS INSTRUMENTS

Printed in U.S.A.

MANUAL REVISION HISTORY

TIFORM Reference Manual (2234391-9701)

Original Issue . . . . . February 1984

The total number of pages in this publication is 364.

The computers, as well as the programs that TI has created to use with them, are tools that can help people better manage the information used in their business; but tools—including TI computers—cannot replace sound judgment nor make the manager's business decisions.

Consequently, TI cannot warrant that its systems are suitable for any specific customer application. The manager must rely on judgment of what is best for his or her business.

Preface

This manual describes the Texas Instruments form-generating software package, TIFORM™. TIFORM provides a comprehensive and consistent means of controlling the input/output (I/O) formats of a video display terminal (VDT) or a keyboard send/receive (KSR) terminal. It permits the application programmer to design applications without regard for the physical characteristics of the VDT. TIFORM also provides editing facilities for input and controls the display characteristics of output. These facilities free the application program from terminal management and provide a high degree of form design flexibility. You can modify or install a form on a different terminal without affecting the data processing application.

A TIFORM form cannot execute by itself. You need to write an application program to drive the form. TIFORM provides a utility, the Form Tester, that allows you to test your form before you write your application program.

The TIFORM user is assumed to be an experienced programmer or someone well acquainted with the concepts and language of data processing. TIFORM interfaces with three procedural languages: COBOL, FORTRAN, and Pascal. The description of the application interface assumes that the reader is familiar with one of these languages.

Designed to be implemented on a Texas Instruments 990/10, 990/12, or Business System minicomputer with a minimum of 256K bytes of memory, TIFORM operates under the DX10 (Release 3.2 or later) and DNOS (Release 1.0 or later) operating systems.

NOTE

This manual applies to both operating systems.

If you are a first-time user, you should read the sections of this manual in order. Section 1 presents the basic concepts of TIFORM. It describes the TIFORM execution environment and the structure of a form. Section 2 describes form execution, including the functions available to the users of various terminal types. Section 3 presents the formal syntax of the Form Definition Language (FDL). You should become familiar with these sections before proceeding.

Section 4 discusses the Interactive Screen Generator/Editor (ISGE). Even though using the ISGE removes the need for a detailed knowledge of the FDL, you still need a working knowledge of FDL concepts and terminology. To specify certain data structures by means of the ISGE, you need an exact knowledge of the syntax of the structures as required by the compiler.

You can test trial forms by using the Form Tester utility. This utility (see Section 7) permits you to experiment with the operation of the form and the keyboard. You must be familiar with the Open Form and Prepare Segment commands, as well as some of the simpler Read and Write commands. Section 5 describes these commands for FORTRAN, COBOL, and Pascal. While testing the form, experiment with the keyboard.

Section 5 describes the functions you need to write an application program that interacts with the form. Read this information carefully. All functions are available to users of FORTRAN, Pascal, and COBOL.

To write an assembly language application, you should use either the COBOL or the Pascal high-level language (HLL) interface packages. Read either the COBOL or Pascal applications interface discussion in Section 5. Refer to the appropriate language manual for the format of the interface from a program written in that language to an assembly language external procedure. Note that the Pascal interface requires the pointers to parameters, with the exception of size parameters, where values are passed.

Before executing an application program that uses TIFORM, you must link that program with the appropriate HLL interface package. Section 6 discusses linking techniques. You can use the link maps in Section 6 as examples.

If you plan to test the application, read the parts of Section 2 appropriate for your terminal type. (Paragraphs 2.1 through 2.7 apply to all terminal types.)

This manual contains a glossary and the following sections and appendixes:

Section

1 General Information — Describes the overall execution environment of TIFORM, including the software components that comprise TIFORM. It describes in detail the capabilities and characteristics of a form and the various structures that make up a form.

2 Form Execution — Discusses the ways in which TIFORM interacts with both the application and the terminal user, and the effects of these interactions on the form. This section discusses the terminal types that TIFORM supports: VDTs, KSRs, sequential files, and printers. This section also discusses edit keys, function keys, and the TIFORM Print key. The Print key allows the terminal user to print the contents of the current screen.

3 Form Definition Language — Discusses in detail the Form Definition Language (FDL), a language that allows you to design flexible, attractive forms for data entry. FDL is a non-procedural, block-structured language that you can use to specify the characteristics of your forms. This section discusses FDL syntax, provides an overview of the functional groups of FDL statements, and then describes each FDL statement in alphabetical order.

4 Interactive Screen Generator/Editor — Describes the Interactive Screen Generator/ Editor (ISGE), a convenient tool that you can use to develop your form interactively from a VDT. When you finish the form, the instructions that the ISGE generates are translated into FDL for access by the driving application.

Section

5 Application Interface — Discusses the interface calls between the application and TIFORM. TIFORM supports applications written in COBOL, FORTRAN, and Pascal. This section discusses all of the interface routines in alphabetical order and presents examples of each call.

6 Linking Application Programs That Use TIFORM — Describes the two techniques for linking your application program with TIFORM.

7 Form Tester — Describes the Form Tester utility, a utility that allows you to test your form before you write an application program to drive it.

Appendix

A Keycap Cross-Reference — Provides cross-references from generic key names to the names on the keycaps of supported terminals.

B TIFORM Status Codes — Describes the status codes that TIFORM returns to the application.

C TIFORM Error Codes — Includes TIFORM error messages, FDL compiler (FDLC) diagnostics, and ISGE error messages.

D Examples of FDL Form Definitions — Presents commented examples of form definitions written in the FDL.

E Graphic Characters — Shows keyboard diagrams for the keys that produce graphic characters on the 911 VDT and the 931 VDT. This appendix also shows the graphic character sets for VDTs.

F Quick Reference to FDL Syntax — Provides a quick reference to FDL syntax. This appendix contains all of the FDL statements and their syntax definitions.

G Quick Reference to the ISGE — Describes the function keys that you can use when designing a segment mask or field mask. This appendix also describes all of the prompts that allow you to specify field attributes.

The following documents contain additional information related to the use of TIFORM with the DX10 and DNOS operating systems:

Title	Part Number
DX10 Operating System Concepts and Facilities (Volume I)	946250-9701
DNOS Concepts and Facilities	2270501-9701
DX10 Operating System Application Programming Guide (Volume III)	946250-9703
DNOS Supervisor Call (SVC) Reference Manual	2270507-9701
DX10 Operating System Error Reporting and Recovery Manual (Volume VI)	946250-9706
DNOS Messages and Codes Reference Manual	2270506-9701
990/99000 Assembly Language Reference Manual	2270509-9701
COBOL Reference Manual	2270518-9701
DX10 COBOL Programmer's Guide	2270521-9701
DNOS COBOL Programmer's Guide	2270516-9701
TI Pascal Reference Manual	2270519-9701
DX10 TI Pascal Programmer's Guide	2270528-9701
DNOS TI Pascal Programmer's Guide	2270517-9701
FORTRAN-78 Reference Manual	2268681-9701
DX10 FORTRAN-78 Programmer's Guide	2268679-9701
DNOS FORTRAN-78 Programmer's Guide	2268680-9701

Contents

2 — Form Execution

3 — Form Definition Language

4 — Interactive Screen Generator/Editor

5 — Application Interface

6 — Linking Application Programs That Use TIFORM

Illustrations

Tables

1

General Information

1.1 INTRODUCTION

TIFORM provides many capabilities designed to simplify data entry and validation for application programs written in COBOL, Pascal, or FORTRAN. When you use TIFORM, your program does not need any special routines for obtaining data from different device types or for handling various combinations of keystrokes. Instead, your program simply calls TIFORM to display an appropriate form, obtain the data, and return the information to your program.

This section describes the capabilities of TIFORM, beginning with a general introduction to its software components. Following a description of the data structures maintained by TIFORM, it explains the attributes you can assign to data fields and the ways you can manipulate data using TIFORM. The section concludes with a discussion of the interface between your application program and TIFORM, intended to help you use TIFORM to your best advantage.

1.1.1 Form Definition Language

The Form Definition Language (FDL) allows you to design flexible, attractive forms for data entry. FDL is a non-procedural, block-structured language that you can use to specify the characteristics of your forms. The FDL compiler translates your FDL statements into a format usable by the Form Executor and installs them as overlays in a specified program file.

A form can consist of one screen display or several. The part of the screen display where the user enters data is called a *segment*. The rest of the screen display consists of directions, prompts, borders, and other constant text and is called the *segment mask*. When you define a segment, you use FDL statements to describe each field on the screen in terms of its size, location, and type of data it accepts. You can also define defaults, initial values, validation tests, and edits for the field.

Here are some additional capabilities available with FDL:

- Grouping of fields with indexing to specific members of the group
- Graphics characters
- Field and cursor blink
- High/low intensity
- Data saved from screen to screen
- Data validation by field and screen
- Support for cursor control and function keys

- Conditional field attributes
- Undisplayed variables
- User-defined error messages and help screens

The Execute FDL Compiler (XFDLC) command allows you to compile your form definitions and store them in a program file as relocatable overlays. XFDLC also produces a listing file containing your FDL source statements and any appropriate warnings or error messages. A separate error file lists only the erroneous statements and error messages.

Each form segment and segment mask is stored as a separate entry in the program file. This allows you to replace a form or segment without having to recompile and relink your application program. Your application program can use multiple forms, and several application programs can use the same form.

1.1.2 Interactive Screen Generator/Editor

The Interactive Screen Generator/Editor (ISGE) is a utility program that helps you design forms for your application. Using ISGE, you can create a new segment (screen display) and modify it until you are satisfied with the layout. ISGE then translates your design into FDL statements and compiles the resulting FDL program. ISGE leads you through the form definition procedure with a series of menus and prompts, using the function keys on your terminal for many common operations.

1.1.3 Form Executor and Interface Routines

The Form Executor is the TIFORM run-time associated with the terminal where your forms are displayed. The Executor receives calls from the TIFORM interface routines linked with your application program and carries out their instructions according to the characteristics of the terminal. TIFORM allows you to tailor forms and segments to specific classes of devices. The Form Executor automatically selects the form or segment that is appropriate for the user's terminal. If you define a form or segment without specifying the device type, the Form Executor uses the same form or segment for all devices.

You can have the Form Executor execute your calls synchronously (completing execution before returning control to your program) or asynchronously (returning control immediately after receiving the call).

The TIFORM interface routines allow your application program to access the forms you have created. By calling these routines, your program can:

- Open, close, change, and reset forms.
- Read and write specified fields, groups, and variables.
- Write with reply—with or without cursor return.
- Write messages to the message line on the terminal.
- Arm and disarm function keys on the terminal.
- Enable and disable a variety of Executor control functions.

The Form Executor usually runs as a shared procedure using different segment overlays for each application task, as shown in Figure 1-1. In this setting, the Executor uses intertask or interprocess communication (depending on your operating system) to interface with the application task. Your application task controls its connection and disconnection with the Form Executor. This allows you to pass an Executor from task to task, circumventing memory limitations. You can also use this facility to allow your task to communicate serially with two or more Executors and their associated terminals.

TIFORM also gives you the option of linking the Executor with your application task. Though this configuration substantially reduces the space available for your program and requires you to link a copy of the Executor with each task that uses TIFORM, it does execute somewhat faster since it does not employ intertask or interprocess communication for the interface. To support this arrangement, TIFORM includes a separate, linkable version of the Form Executor and a set of interface routines designed for use with the linkable Executor. These are shipped on the installation media, but are not installed automatically because of the relative infrequency of their use.

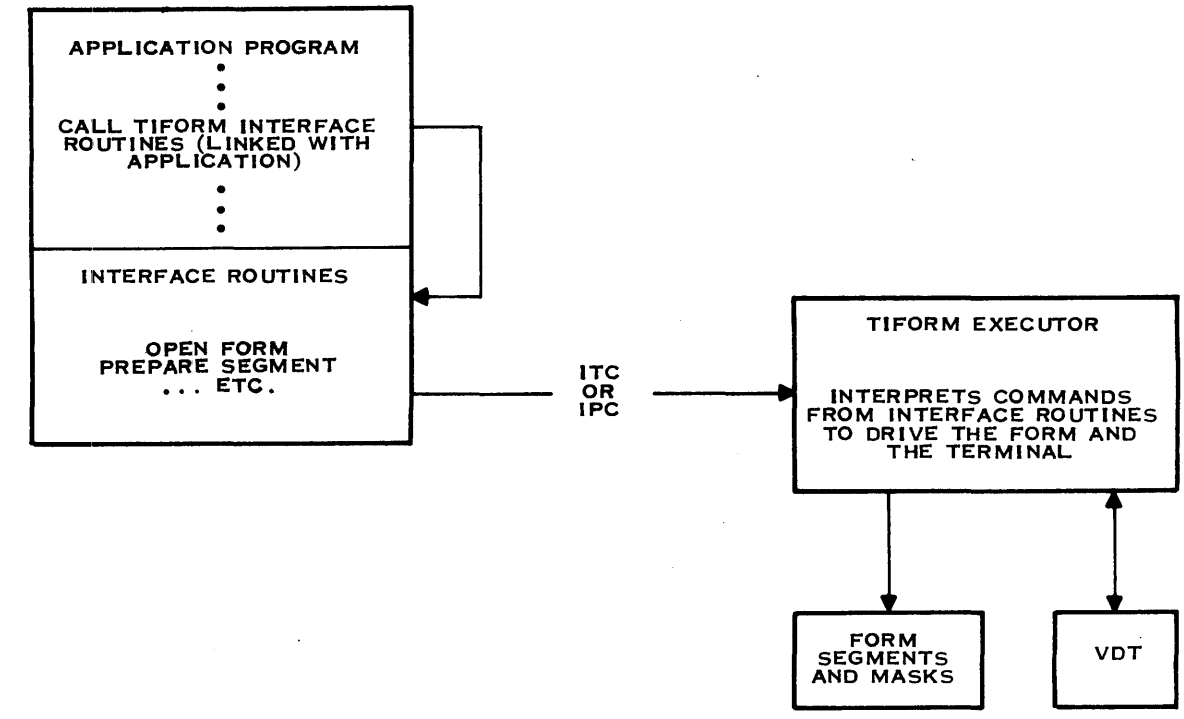


Figure 1-1. TIFORM Execution Environment

1.1.4 Form Tester

The Form Tester is a utility that enables you to test your forms without having to write a test program. The Form Tester serves as an interactive driver program that allows you to simulate calls to the interface routines and immediately see their results. The 22 functions built into the Form Tester can exercise virtually all of the capabilities of TIFORM. Each function produces a status display that includes all of the information that the corresponding interface routine would pass to your application program as return parameters or items in the TIFORM status block.

Using the Form Tester is a good way to familiarize yourself with the capabilities of TIFORM. For more information on the Form Tester, please refer to Section 7.

1.2 FORM COMPONENTS

A *form* consists of segments, fields, and variables. A form can span one or more screens or pages with a different segment for each screen or page. *Fields* are places shown in the segment where the user can enter data, while *variables* are hidden and used for intermediate results and calculations. Fields have defined characteristics, called attributes, that determine how they are displayed and what kind of data they accept. Field attributes are discussed in detail elsewhere in this section of the manual. Figure 1-2 shows an example of a TIFORM segment.

TIFORM also supports field arrays and groups. A *field array* is a matrix of identical fields arranged in rows and columns. A *group* is a collection of fields, variables, and subgroups that are given a common name. Read and write routines enable you to use an index to reference individual members of a group.

1.2.1 Form

To the user of a TIFORM application, a form is a sequence of screen displays used to enter data for the application. To the application program, a form is a collection of segments, previously defined in FDL and stored together in a program file. The FDL FORM statement assigns a name to the form, which becomes the name of the form root overlay. The application program indicates which form to use by specifying the name of the form and its program file in a call to the Open Form routine. This is the only use of the form name by the application.

For each form, you can specify a fill character and a device type. If you specify a *fill character*, that character replaces the blank as the filler for all input fields on the form. You can override this fill character for a particular segment or field, if you wish. *Device type* refers to the generic device where the form is displayed. TIFORM supports several types of video display terminals (VDTs) and keyboard send/receive (KSR) devices.

1.2.2 Segment

A *segment* is a part of a form displayed for data entry. The segment has fields where the user can enter data and where the program can display messages. The segment can also have a *segment mask*, which consists of graphics or text that is displayed with the segment but not used for data entry. When a form has more than one segment, you can think of the segment as a page or screen of information belonging to that form. There is no practical limit to the number of segments a form can contain. A segment can belong to up to 10 different forms.

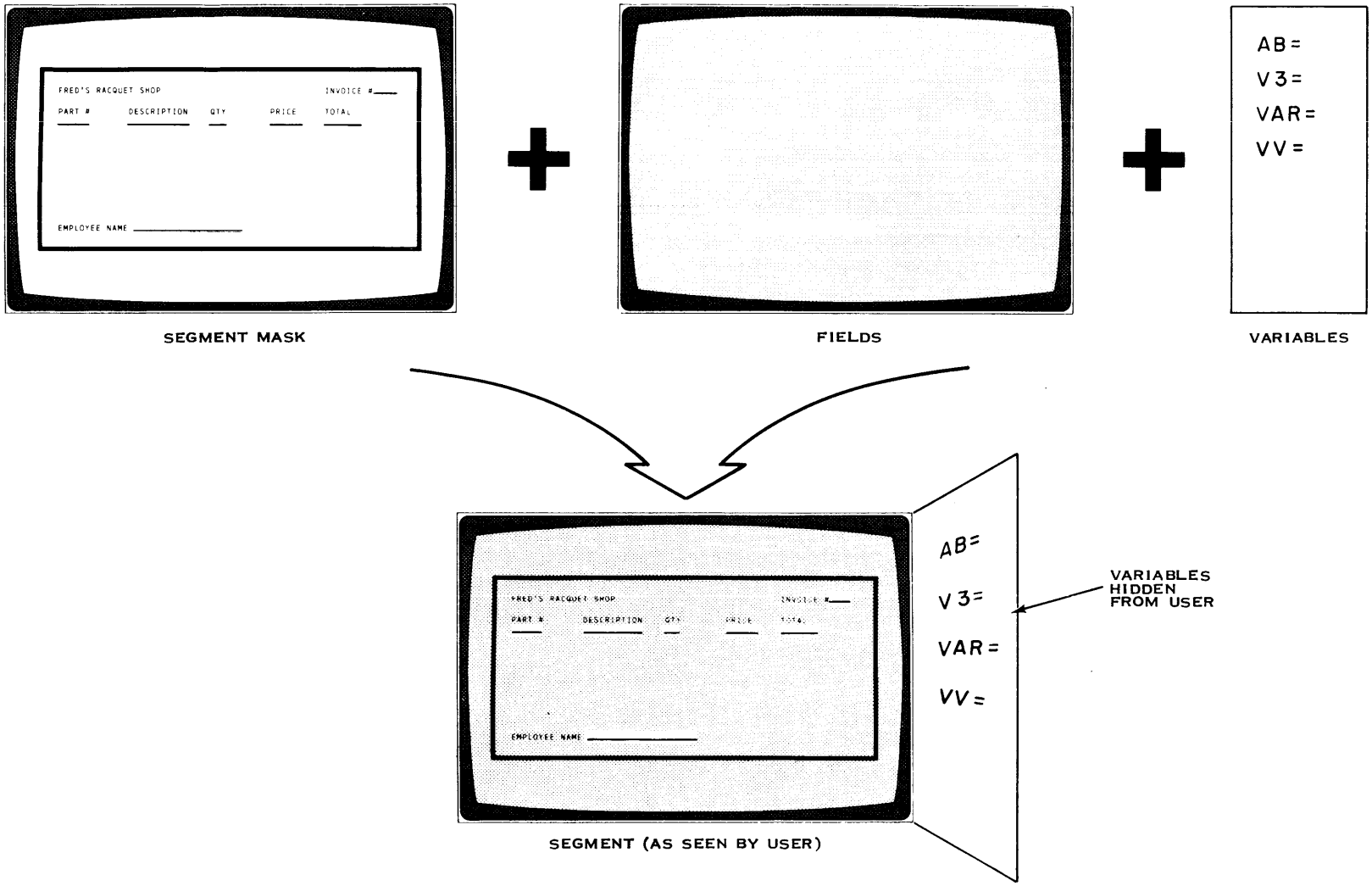


Figure 1-2. TIFORM Segment

Though most applications use one segment and its mask to present the user with an entire page or screen of data, this is not required by TIFORM. In the strictest sense, a segment is simply a collection of fields and variables. The segment mask is optional and does not even have to cover the entire screen or page. Displaying a new segment mask does not require erasing any previous display. Since you can define fields outside the portion of the screen covered by the segment mask, it is possible to implement split screens and windows as part of your application.

Using FDL or ISGE, you define the characteristics of the segments and segment masks used in your application. You must compile a segment and its mask together and store them in the same program file. After opening a form, your program can display any of the segments associated with that form by specifying the name of the segment in a Prepare Segment call. The Prepare Segment call brings the segment into memory where your program can issue calls to do reads, writes, and other operations on the segment.

You can assign a specific device type to a segment by including a DEVICE statement in its definition. The DEVICE statement appends a device code to the segment name in the program file, restricting the use of that segment to devices of the type specified. Your program does not need to include the device code in its calls. The Form Executor automatically selects a segment that is appropriate for the device in use. If you do not assign the segment a device type, the FDL compiler creates a version that can be displayed on any supported device.

Also within a segment, you can define edit sets, conditions, validation lists, and error messages. *Edit sets* define transformations to be performed on data entered (such as changing Y to YES). *Conditions* are logical expressions that state requirements that the data must satisfy before it is accepted by the application. *Validation lists* identify the acceptable values for fields, while *error messages* provide the appropriate text to display when the user attempts to enter an invalid value. A subsequent paragraph on data manipulation discusses these capabilities in greater detail.

1.2.3 Field

A field is part of a segment used for I/O between the application program and the user. Each field occupies one or more consecutive horizontal positions on a segment, as determined by FDL statements for the field. Using calls to the interface routines, the application program can read and write information in the fields for the currently active segment. *Output only* fields can only be written by the application program—they cannot be used for data entry. *Input/Output* or *I/O* fields can be both written and read. For instance, an application program can first write a dollar sign into a monetary I/O field and later read the user's response from the same field.

The length, location, and usage of a field are only three of its defined characteristics, called *attributes*. FDL allows you to specify additional attributes that determine how the field is displayed, what data it accepts, and how the cursor behaves when it leaves the field. For output-only fields, you can supply only the position and display attributes, since output-only fields are displayed exactly the way they arrive from your program. I/O fields can have additional attributes for editing and processing the data entered into them. Each is briefly described in a subsequent paragraph in this section.

In the FDL for a segment, the definition of a field begins with a FIELD statement that assigns it a name. The FIELD statement also marks the beginning of a *field block*, a series of FDL statements that define the field and its attributes. The field name consists of a letter followed by up to five additional letters, numerals, dashes (–), and dollar signs ($). To make the name available to your program, you must list it in an EXTERNAL statement in the FDL for the field or its segment. Each field name must be unique within its segment.

Like segments, I/O fields can have their own masks, which are loaded into memory at the same time as the segment. The *field mask* consists of background text to be displayed in a constant or relative position. If a field's definition includes a DISPLAY MASK statement, the indicated field mask is displayed whenever the cursor enters the field. By including a POSTCLEAR specification in the DISPLAY MASK statement, you can have the mask erased when the cursor leaves the field.

KSR devices present fields to the user one at a time, since they don't have screens to display an entire segment at once or cursors to show where to enter data. To allow you to provide additional information telling the KSR-type device user what to enter, a field block can include a PROMPT statement to specify the text to print each time the user is to enter data into the field.

1.2.4 Variable

Variables are like fields, except that they are not displayed. They give you a way to store data with the segment for such uses as defaults, comparisons, or calculations. To define a variable, you include a VARIABLE statement in the FDL for a segment, giving the name and initial value for the variable. The name must be unique within the segment and consists of a letter followed by up to five additional letters, numerals, dashes (–), and dollar signs ($). To make the name available to your program, you can list it in an EXTERNAL statement in the FDL for the field or its segment. The initial value is a character string whose length determines the size of the variable.

1.2.5 Array

An array is a collection of fields arranged in rows and columns. All fields in an array have the same attributes, except for location. You define an array in FDL by defining a field and including an ARRAY statement in its specifications. The name you assign the field becomes the name of the array, and any attributes you assign the field are shared by all elements of the array. Like other field names, array names must be unique within the segment and consist of letters, numerals, dashes, and dollar signs following an initial letter. However, an array name must be short enough to include an index within its six-character length. Arrays of up to 9 elements need a one-character index, arrays of 10 through 99 elements need a two-character index, and so on.

To reference a particular field within an array, you give the name of the array, followed by its row and column. If you supply an asterisk instead of a row or column number, it is understood that you mean all rows or all columns in the array.

1.2.6 Group

A group is a named list of fields, variables, arrays, and other groups. You define groups in FDL using either the GROUP or ORDERED GROUP statement. The name must be unique within the segment and consists of a letter followed by up to five additional letters, numerals, dashes (–), and dollar signs ($). You do not need to use an EXTERNAL statement to make the group name available to your program. The FDL compiler does so automatically. Also, you do not need to .define a group that contains the entire segment. The segment itself serves as a group having the same name.

If you use a GROUP statement to define a group, you accept the screen-order sequence of items in the group. That is, the items in the group are processed in the order they appear on the screen—left to right, top to bottom. Subgroups listed as members of the group are replaced by the items they contain.

If you use an ORDERED GROUP statement to define a group, you establish a new sequence among the items it contains. Fields, variables, and arrays are kept in the same sequence in which you list them in the FDL. Subgroups on the list are replaced in sequence by the items they contain and in the order established for the subgroup.

Subgroups can be nested 10 deep. The type of the subgroup determines the order of the fields, variables, and subgroups it contains. For example, if an ordered subgroup contains an unordered subgroup, the items in the ordered group are processed in the order they are listed in the group's definition. When the processing reaches the unordered subgroup, its items are processed in screen order. Following the last item in the unordered subgroup, processing reverts to the defined order for the ordered group.

Groups define the scope of read and write operations performed by the interface routines. When your program calls one of the Read routines, it indicates the group that contains the items wanted. The data returned depends on the Read routine used, as follows:

- Group Read — Your program receives the data from all I/O fields in the group (and in all subgroups it contains).

- Indexed Read — The call uses an index to specify an item on the list where the Read begins and a counter to specify how many items to read. The call can include a second index to handle situations in which an item in the group is itself a group. This second index allows you to indicate a specific item within the subgroup. If that item also turns out to be a group, the Read begins with its first member—there are no further layers of indexing.

Group Write and Indexed Write routines work the same way, except that they write data to the indicated items instead of reading it. A subsequent paragraph provides details on the format of the data read and written, plus a closer look at the interface routines available.

1.3 FIELD ATTRIBUTES

The attributes you assign to a field determine how it is displayed to the user, which kinds of data it accepts, and what processing follows data entry. Each set of attributes is called an *edit set*. Usually, you define an edit set for each field by supplying the appropriate FDL statements as part of its field block. The following paragraphs introduce each of the field attributes you can define with FDL. Refer to Section 3 for details on the individual FDL statements involved.

Sometimes an application requires a field to have two or more edit sets with prevailing conditions determining which one to use. In this case, you define the conditions as part of the field block and each edit set in a separate *edit set block*. The EDIT SET statement marks the beginning of the edit set block and assigns it a name. Within the edit set block, you can define alternative attributes for the field using any of the FDL statements available for the field block except for ARRAY, DEFAULT, EXTERNAL, OUTPUT, and POSITION.

You can define conditions according to the character composition, length, or value of the data at the time the user enters data into the field or before the data is returned to your program. The condition can determine which edit set to apply or where to move the cursor for the next operation. The *condition block* begins with a CONDITION statement, which supplies a name for the condition, and ends with an END CONDITION statement. (Like most TIFORM names, an edit set or condition name must begin with a letter and can include up to five additional letters, numbers, dashes, and dollar signs.)

For conditional edits, FDL provides IF-THEN-ELSE logic and a NOT operator. The IF statement can appear in either a field block or an edit set block. Likewise, most of the validation tests described for the field block are also available as conditions in the condition block. Many conditions involve lists, such as a list of characters that make up a valid entry. The condition block contains only a reference to the list, while the list itself must be in the segment block but outside any field, edit set, or condition block.

1.3.1 Position

Every field must have a *position* attribute that defines how many characters it can contain and where it begins on the screen or page. The only limit to the length of a field is the length of the line where it appears, since fields cannot be continued from one line to the next. Otherwise, a field can begin and end anywhere you like. Fields defined on the bottom line of a terminal screen can be temporarily lost when an error message is displayed. The fields are restored when the message is cleared and the cursor is positioned in a field on the bottom line.

You define a field's position by including a POSITION statement in its field block. You can specify the location of the first character in the field either by indicating its row and column or by giving its row and column displacement from a location previously specified in a POSITION or background text (M) statement. In either case, you also need to specify the length of the field.

1.3.2 Output and No Entry

The *output* and *no entry* attributes allow you to display a field without permitting the user to enter data into it. Unless you assign the field an output or no entry attribute, the user is free to enter or modify data in the field. To assign one of these attributes to a field, you include an OUTPUT or NO ENTRY statement in its field block. The output and no-entry attributes differ in the way the field is processed and the other attributes it can have.

If you assign the output attribute to a field, it becomes an output field and the only other FDL statements allowed in its field block are POSITION, DISPLAY, and ARRAY. Your program can read, write, and reset the field, but any read always returns blanks.

If you assign the no-entry attribute to a field, it remains an I/O field with all of the properties of an I/O field except that the cursor can never enter the field during a read. Post-entry processing occurs as usual and Read routines return the appropriate values.

1.3.3 Initial Value

When a field has an *initial value* attribute, it automatically receives a specified value whenever it becomes the current field. As the cursor enters the field, the initial value is displayed, making it easy for the user to enter that value by simply tabbing through the field. If you do not assign an initial value to the field, its current value is displayed whenever the cursor enters the field.

To assign an initial value for a field, you include a VALUE statement in its field block, specifying a literal, variable, or another field as the initial value. If you specify a literal, that value is displayed in the field whenever the cursor enters it. If you specify a variable or a field, the current value of the variable or field is displayed.

1.3.4 Default Value

When a field has a *default value* attribute, it automatically receives a value at the time the segment is displayed. If the cursor never enters the field, this value is also used in post-processing. The default value is different from the initial value in that the cursor must enter the field for the initial value to come into effect and in that the initial value is restored each time the cursor returns. The default value is only displayed once—when the Prepare Segment routine is called. (The default value is also different from default responses provided by some System Command Interpreter (SCI) commands. Unlike SCI defaults, TIFORM default values are *not* returned to the application when the user erases the contents of the field.) If you do not assign a default value to a field, a zero-length value is provided when the segment is displayed.

To assign a default value to a field, you include a DEFAULT statement in its field block, specifying a literal, variable, or another field. If you specify a literal, that value is inserted into the field when the screen is displayed. If you specify a variable or a field, the current value of the field or variable is displayed.

1.3.5 Required

A field with a *required* attribute cannot be left empty by the user. If a field does not have the required attribute, the user can skip it or erase any value already there. To assign the required attribute to a field, you include a REQUIRED statement in the FDL for its field block. To remove the required attribute, you use a NOTREQUIRED statement. If the field block does not have either a REQUIRED or NOTREQUIRED statement, the required attribute is not assigned.

If a field has a required attribute and the user attempts to move the cursor out of the field without entering any data, an error message is displayed at the bottom of the screen. When the user acknowledges the message, the cursor returns to the beginning of the field. If you like, you can provide your own message text to replace the standard message by using the optional DIAGNOSTIC portion of the REQUIRED or NOTREQUIRED statement. You can either specify the new message text or the name of an error message defined in the segment block.

If a field does not have a required attribute, the user can enter a zero-length response, even if other attributes of the field require an entry of some specified minimum or exact length. To require the user to enter a certain number of characters into the field, you must assign it the required attribute as well as an appropriate minimum or exact length attribute.

1.3.6 Minimum Length

When the user enters data into a field with a *minimum length* attribute, the entry is rejected unless it contains at least a minimum number of characters. To assign a minimum length attribute to a field, you include a MINIMUM LENGTH statement in its field block. In the MINIMUM LENGTH statement, you specify the minimum number of characters in a valid entry and, optionally, the text or name of an error message to appear at the bottom of the screen when the user attempts to enter a response with fewer characters than the minimum.

The minimum length must be greater than zero and less than or equal to the length of the field. If you want to force the user to enter data until the field is full, you can set the minimum length equal to the length of the field. To prevent the user from skipping the field (or giving a zero-length response), you should also assign the required attribute to the field.

1.3.7 Exact Length

When the user enters data into a field with an *exact length* attribute, the entry is rejected unless it contains a specific number of characters. To assign an exact length attribute to a field, you include a LENGTH LIST statement in its field block. You can use the optional DIAGNOSTIC specification to supply a replacement for the standard error message displayed when the user attempts to enter a value with some other length.

Elsewhere in the segment, you need a companion LIST LENGTH statement that lists the valid length or lengths for data entered into the field. The length of the data entered must match one of the lengths on the list. To prevent the user from skipping the field (or giving a zero-length response), you should also assign the required attribute to the field.

1.3.8 Value Range

The *value range* attribute applies only to numeric data. If a field has a value range attribute, it accepts only values that lie within a defined range or set of ranges. A user who attempts to enter an unacceptable value receives an error message and must try again. When the user enters a number into a field with a value range attribute, the number is compared to a list of valid ranges assigned to the field. If you define the list as *inclusive*, the value is accepted only if it falls within one of the ranges on the list. (Matching the upper or lower bound of a range also satisfies the requirement.) If you define the list as *exclusive*, the value is accepted only if it falls outside of every range on the list.

To assign the value range attribute to a field, you include a LIST RANGE statement in its field block and a companion RANGE LIST statement elsewhere in the segment block. The LIST RANGE statement gives the name of the range list to use and, optionally, a replacement for the standard error message text. If you want to define an open-ended range of values, you must use a PASS/FAIL statement instead of LIST RANGE.

1.3.9 Value Table

If a field has a *value table* attribute, it accepts only values that appear on a table of valid values. When the user enters data into a field with a value table attribute, the value entered is compared to a list of valid ranges assigned to the field. If the value does not appear on the table, the user receives an error message and must try again.

To assign the value table attribute to a field, you include a LIST TABLE statement in its field block and a companion TABLE LIST statement elsewhere in the segment block. The LIST TABLE statement gives the name of the table list to use and, optionally, a replacement for the standard error message text. If you define the list as *inclusive*, the value is accepted only if it appears on the list. If you define the list as *exclusive*, the value is accepted only if it does *not* appear on the list.

1.3.10 Value Comparison

A field with a *value comparison* attribute requires data to pass a comparison test before it is accepted. The comparison test checks the relationship between the current values of two fields or variables. Neither has to be the field that has the value comparison attribute. When the user enters data into the field, the comparison test is made. Depending on whether the data entered passes or fails the test, processing continues or the user receives an error message and tries again.

To assign a value comparison attribute to a field, you include a PASS/FAIL statement in the FDL for the field block. The PASS/FAIL statement consists of your choice of the keywords PASS or FAIL and an IF specification that states a simple relational condition. The operands in the condition are names of fields or variables. (As a convenience, an asterisk (*) represents the current field.) The operator is one of the relational operators: EQ, NE, LE, LT, GE, or GT. For comparisons involving nonnumeric fields or variables, only the EQ and NE operators are available.

1.3.11 Character Set

The *character set* attribute determines the set of valid characters that the user can enter into the field. To assign a character set attribute to a field, you include a CHARACTER LIST statement in its field block and a LIST CHARACTER statement elsewhere in the segment block. The CHARACTER LIST statement gives the name of the companion LIST CHARACTER statement and optional diagnostic text to replace the standard error message. The LIST CHARACTER statement provides a list of valid characters or ranges of valid characters.

1.3.12 Numeric

Fields with the *numeric* attribute accept only numeric data. The user can enter only the numerals 0 – 9 and in some cases blanks, a sign, and a decimal point. Numeric fields are right-justified before they are returned to the application program.

When you include a NUMERIC statement in a field block, the FDL compiler generates attributes for the field that are equivalent to those provided by the JUSTIFY, SCALE, LIST CHARACTER, and CHARACTER LIST statements. The NUMERIC statement allows you to specify a sign, fill character, and decimal point for the field.

1.3.13 Tabstop

If a field has the *tabstop* attribute, the cursor stops at the beginning of the field whenever the user presses the Forward Tab key to exit the previous field. If the user presses the Forward Tab key and there are no fields with the tabstop attribute left to read, the cursor simply moves to the next field. To assign the tabstop attribute to a field, you include a TAB statement in its field block.

NOTE

Throughout this manual, the names of the keys are generic key names unless referring to a specific terminal such as the 931 VDT. Then the specific key name is used. In some cases, the names on the keycaps of the terminals match the generic key names, but in many cases they do not. Appendix A contains a table of key equivalents to identify the specific keys on the terminal you are using. Drawings that show the layout of the keyboard of each type of terminal are also included.

1.3.14 Autoskip

If a field has the *autoskip* attribute, the cursor automatically moves to the next field after the user enters a character into the last position in the field. Otherwise, the user must close the field by pressing the Next Field, Skip, Forward Tab, Enter, or Return key. To assign the autoskip attribute to a field, you include an AUTOSKIP statement in its FDL.

1.3.15 Graphics Input

If a field has the *graphics input* attribute, the user can enter graphics characters as data. To assign the graphics input attribute to a field, you include a GRAPHICS INPUT statement in its field block.

1.3.16 Display

A field's *display* attributes determine how values in the field appear on the user's screen. The options available depend on the capabilities of the terminal, as summarized in Table 1-1. TIFORM can support the following display attributes:

Attribute	Meaning
BR	Bright — Characters are displayed in high-intensity.
BL	Blink — Characters (or cursor) blinks.
GR	Graphics — Virtual graphic characters become true graphics characters when displayed.
ND	Nondisplay — Characters are not displayed on the screen.

The display attributes also apply to segment masks, field masks, and edit sets.

Table 1-1. Display Attributes of Supported Terminals

Attributes	911	931	940 *	820
Bright intensity	YES	YES	YES	NO
Cursor/Field blink	YES	YES	YES	NO
Virtual graphics	YES	YES	YES	NO
Nondisplay	YES	YES	YES	YES

Note:

* Includes Business System 300 Computer and Terminals

1.3.17 Scaling

The *scaling* attribute calls for the Form Executor to multiply or divide the value of a numeric entry by a power of ten. When the user enters a number without a decimal point into a field with a *left* scaling attribute, a decimal point is inserted a specified number of positions from the right end of the field—in effect dividing the number by a power of ten. When the user enters a number without a decimal point into a field with *right* scaling, a specified number of zeros are inserted at the end of the field—in effect multiplying the number by a power of ten. If a field has both a scaling attribute and a justification

To assign a scaling attribute to a field, include a SCALE statement in its field block. If you include an ON COMPLETION specification in the SCALE statement, the scaling occurs when the data is returned to your program and the user does not see the result. Otherwise, the scaling occurs on entry and the user sees the result as soon as the cursor leaves the field. Entering a valid decimal point overrides scaling on entry but not on completion.

1.3.18 Justification

A *justification* attribute calls for the Form Executor to left or right justify data that does not fill the entire field, as follows:

- Left justification — Usually used for text data where the application program expects to receive a character string of a certain length, without leading blanks but with enough trailing blanks to fill the field. In this case, you assign the field a left justification attribute by including a JUSTIFY statementin its field block, specifying left justification and the blank as its fill character.

- Right justification — Usually used for numeric data where the application program expects to receive a valid numeric value, possibly with a sign to the left of the first significant digit or as the rightmost character in the value. In this case, you assign the field a right justification attribute by including a JUSTIFY statement in its field block, specifying right justification, a zero or blank fill character, and an optional number of digits to the right of the decimal point.

The JUSTIFY statement allows you the option of justification ON ENTRY or ON COMPLETION. If you specify ON ENTRY, justification occurs as the user enters data, displaying the justified value immediately after the cursor leaves the field. If you specify ON COMPLETION, the justification occurs just before the data is returned to your application program and the user never sees the result. If you specify right-justification for a numeric field and your application program is written in COBOL, the sign is moved to the rightmost position in the field before the value is returned to your program.

1.3.19 Substitute

When the user enters data into a field with a *substitute* attribute, that value is replaced by a predetermined substitute. To assign the substitute attribute to a field, you include a SUBSTITUTE LIST statement in its field block and a LIST SUBSTITUTE statement elsewhere in its segment block. In the SUBSTITUTE LIST statement, you give the name of the LIST SUBSTITUTE statement and specify whether the substitution is to take place ON ENTRY or ON COMPLETION. In the LIST SUBSTITUTE statement, you define a substitution list consisting of pairs of values and their substitutes. If the value does not appear in the substitution list, it remains unchanged.

1.3.20 Copy

A *copy* attribute causes a specified field, value, or literal to be copied to another field or variable after the user enters data into the field that has the attribute. Though the field with the copy attribute is usually the one copied, this is not a requirement.

To assign the copy attribute to a field, you include a COPY statement in its field block. Like several other attributes, you can specify whether the operation takes place ON ENTRY or ON COMPLETION. If you choose ON ENTRY, the copying occurs immediately after the user enters data into the field and the copy can be made to any field or variable in the segment. If you choose ON COMPLETION, the copying does not occur until the data is returned to your program and the copy must be made to the field with the copy attribute.

1.3.21 Branch and Terminate

The *branch* and *terminate* attributes determine what action follows the entry of data into a field with one of these attributes. Without these attributes, the user enters data into fields on the screen in left-to-right, top-to-bottom order in an unordered group read. In an ordered group read, the user enters data according to the order of the group. After the user enters data into the last field in the group being read, the data in all fields in the group is revalidated, any on-completion processing takes place, and the data is returned to your program.

When you assign a branch attribute to a field by including a BRANCH statement in its field block, you specify the next field where the user is to enter data. Unless that field also has a branch attribute, the cursor moves to subsequent fields in the usual order. You assign the terminate attribute to a field by including a TERMINATE READ statement in its field block. If you specify that the termination is to occur IMMEDIATELY, all data is returned to your program without the usual revalidation or on-completion processing. If you do not specify an immediate termination, revalidation and on-completion processing occurs but the Read terminates before the user has a chance to enter data in any other fields.

You can assign *conditional branching* and *conditional termination* attributes to a field by including an IF CONDITION statement in its field block, specifying the name of a condition block defined elsewhere in the segment. You must also include a THEN specification that specifies the next action to take when the condition is true and/or an ELSE specification to indicate what to do when the condition is false. For conditional branching, you include a GOTO specification with the name of the next field to process. For conditional termination, you include a TERMINATE or TERMINATE IMMEDIATELY specification following THEN or ELSE.

Though the condition usually involves a test on the field with the conditional branch attribute, this is not a requirement. The condition can test any field or variable in the segment. If a field block contains more than one IF CONDITION statement, the tests are made in the order of the statements in the block. As soon as a test succeeds for a conditional branch or terminates with a THEN specification—or fails for one with an ELSE specification—the branch or termination occurs and no further conditions are tested.

1.3.22 Novalidate

Occasionally, you want to make an exception to usual data validation requirements and have the contents of a field returned to your program even though the data does not satisfy all of its edits at that time. If you assign a *novalidate* attribute to a field by including a NOVALIDATE statement in its field block, you exempt it from the usual revalidation that occurs for all fields just prior to returning the data to your program. The novalidate attribute does not exempt the field from tests made at the time the user enters the data or from any other post-processing that occurs at the end of the Read. Its main use is to allow on-entry substitutions to take place without requiring the substitute values to satisfy all of the same conditions as the ones originally entered.

1.4 APPLICATION INTERFACE

An application program that uses TIFORM can read and write data to the user's screen in much the same way as it reads and writes data to a file. All the program has to do is call one of the interface routines, specifying a buffer for the data being read or written. The TIFORM status block returns information on the operation's success or failure and the operating system status. For read operations, it also returns the key the user pressed to terminate the operation and the location of the cursor at the end of the read.

TIFORM handles the difficult part of screen I/O. It controls the movement of the cursor, carries out edit key functions, and makes sure the data entered by the user is acceptable to the application program. The following paragraphs describe this exchange of information from the point of view of the application program.

1.4.1 Data Buffer

Instead of using COBOL, Pascal, or FORTRAN commands to read and write data to the screen, TIFORM applications use data buffers and the interface routines. The data buffer consists of consecutive bytes of character data as represented in the host language:

- Programs written in COBOL 3.1 (or earlier versions) use a 01-level data item with PICTURE X(n), where n is the number of characters in the buffer. Since the COBOL 3.1 (CF$) interface routines cannot tell the size of the data area, the program should declare another data item with PICTURE X immediately after the buffer. This allows the program calling the COBOL 3.1 interface routines to specify the beginning and end of the data buffer.

- Programs written in COBOL 3.2 (or later versions) can use any data item with PICTURE X(n), where n is the number of characters in the buffer. Since COBOL 3.2 does not require parameters to begin on a word boundary, the data buffer can be a data item of any level. Also since COBOL 3.2 includes run-time routines that enable it to determine the size of a data area, the COBOL 3.2 (CX$) interface routines do not need another data item to indicate the end of the data buffer.

- A Pascal program uses a variable PACKED ARRAY of CHAR for a data buffer. TIFORM provides two sets of interface routines for Pascal. The EXTERNAL FORTRAN (PF$) routines require the application program to pass the beginning and end of the array as parameters in the call. Therefore, a program using the EXTERNAL FORTRAN routines defines a data buffer as a record consisting of the PACKED ARRAY followed by a simple INTEGER variable. The EXTERNAL (PX$) interface routines allow the program to take advantage of the Pascal upper bound (UB) function. These routines expect to receive the name of the PACKED ARRAY and its size, which is easily expressed with UB. The data buffer is simply the PACKED ARRAY [1..?] OF CHAR.

- A FORTRAN program uses an array of CHARACTER*2 for a data buffer. Because the FORTRAN (FF) interface routines expect to receive the beginning and end of the array as parameters, the array should have one more element than the number of words being read or written.

1.4.2 Fields and Variables

All of the fields and variables in a form can be read and written as part of the group to which they belong. Fields and variables that have the EXTERNAL attribute can also be read or written individually. Though all of the interface routines that read and write groups can also handle individual fields and variables, this can require additional memory.

All external names used in a segment are placed in a run-time name table, which is kept in memory while the segment is active. To minimize the memory required for the segment, the form definition should assign the EXTERNAL attribute only to those fields and variables that the application program plans to read and write individually. The application program should perform reads and writes at the group level whenever practical, using group indexing to limit the scope of the operation.

1.4.3 Read Operations

The interface routines include three types of read operations:

- Read Group — Reads values from the form segment for all fields and variables in a specified group. When a group contains subgroups, the fields and variables in those subgroups are also included in the read.

- Read Indexed — Reads the data for a specified number of fields and variables in a specified group, beginning at a specified field, variable, or subgroup. The operation can involve one or two indexes. The first index indicates the field, variable, or subgroup where the read begins. If the first index indicates a subgroup, the second index can identify a member of the subgroup where the read begins.

- Read Indexed With Cursor Return — Performs the same operation as Read Indexed, except that the routine returns index parameters that indicate the position of the cursor when the user terminates the read by pressing an armed event key or the Enter key.

All three routines return the data read to the data buffer specified in the call. The value returned for each field and variable reflects any post-entry processing specified by its FDL, including the insertion of the appropriate fill characters to bring the length of the value up to the length of the field or variable. (The length of a variable is the length of its initial value.) The values are concatenated and returned to the application program in the data buffer as character data. The program must extract the individual values and perform any necessary type conversions.

The order of values in the data buffer depends on the type of group being read. Values from ordered groups are returned in the order defined in the FDL of the group. Values from unordered groups are arranged in screen order, left to right and top to bottom. When a group contains a subgroup, the type of the subgroup determines the order of the values within the subgroup.

1.4.4 Write Operations

The interface routines include four types of write operations:

- Write Group — Writes values from the data buffer to the screen for all fields and variables in a specified group. When a group contains subgroups, the fields and variables in those subgroups are also included in the operation.

- Write Indexed — Writes the values for a specified number of fields and variables in a specified group, beginning at a specified field, variable, or subgroup. Like Read Indexed, the Write Indexed operation can involve one or two indexes. The first index indicates the field, variable, or subgroup where the write begins. If the first index indicates a subgroup, the second index can identify a member of the subgroup where the write begins.

- Write Indexed With Reply — Writes values from a data buffer to the screen in the same way as Write Indexed, and then reads values from the same fields and groups into a data buffer in the same way as Read Indexed. The same data buffer can be used for both parts of the operation.

- Write Indexed With Reply and Cursor Return — Performs the same operation as Write Indexed With Reply, except that the second part of the operation is a Read Indexed With Reply instead of Read Indexed. On completion, the indexing parameters indicate the position of the cursor when the user terminates the reply by pressing an armed event key or the Enter key.

All of the write routines move data from a data buffer to the active form segment and therefore to the user's screen. The write routines regard the data buffer as a concatenation of the values to be used in the write, arranged in the order they are to be written. For ordered groups, the values are taken from the buffer in the order defined for the group. For unordered groups, the values are assigned to I/O fields in screen order, left to right and top to bottom.

To prepare for a write, the application program moves values into the data buffer, adding the necessary fill characters to make their lengths match the lengths of the fields that are to receive them.

1.4.5 Function Keys

A particularly useful feature of TIFORM is the flexibility it provides for the use of function keys. Particular function keys are described with the various terminal types, but some general types are worth noting here:

- Assigned function keys — Function keys associated with fields by FKEYS statements in FDL. When the user presses an assigned function key, the cursor moves to the field associated with it.

- Event keys — Function keys that are armed and disarmed by the application program. The user can terminate a read by pressing an armed event key, even if all of the data has not been entered. Before the application program receives the data, all fields are validated and post-processed according to their attributes.

- Abort keys — Subtype of event keys that are also armed and disarmed by the application program. Unlike other event keys, when the user presses an armed abort key, the current Read is terminated immediately, without any field validation or post-processing.

- Print key — Special feature of TIFORM. When the user presses the Print key, the Form Executor sends a copy of the current contents of the screen to a designated file or printer. A file in the S$TIFORM directory contains a list of terminals and their associated Print key destinations, allowing the screen image to be sent to a specified printer or directory, the default printer LP01, or the TIFORM print queue. Once the application program arms the Print key, it becomes an ordinary event key and no longer performs the print screen function.

1.4.6 Return Status

Each interface routine reports the success or failure of its operation by posting a return code in the status block. The application program should check this code following every call to the interface routines. Otherwise, error conditions can go undetected. The program should take into account that some of the nonzero codes indicate noteworthy, but not necessarily erroneous, conditions.

1.4.7 Control Modes

Since some applications require slight variations on the way the Form Executor usually operates, TIFORM allows the form definition and application program to invoke various control modes. Table 3-1 provides a list of control modes and how they affect normal form execution. Taking advantage of these control modes can free an application program from doing special processing for these situations.

2

Form Execution

2.1 INTRODUCTION

The TIFORM Form Executor interacts with both the application program and the terminal user. This section discusses these interactions, including the following:

- Status information returned from each TIFORM command
- Edit keys
- Function keys
- Terminal devices
- Sequential files and printers
- Print key

2.2 APPLICATION INTERACTIONS

The application interacts with the Form Executor by calling the interface routines. The application is in control. Upon completion, each command returns two status fields in the application program's status block (see paragraph 5.4). Each status field consists of two ASCII characters.

The form status field is a two-digit, decimal field. Its values indicate whether the Form Executor successfully executed the command, as follows:

Status	Meaning
0	Successful execution
1 – 9	Special, nonfatal condition occurred on execution of the command
10 or more	Fatal error

The I/O status field is a two-character, alphanumeric field. If the error resulted from a supervisor call (SVC), this field contains the hexadecimal SVC status. Otherwise, this field is set to ASCII zeros. Note that since the I/O status field may contain a two-character hexadecimal code, you must declare it in COBOL as PIC XX, not PIC 99.

Refer to Appendix B for a listing of the possible TIFORM form status codes and their meanings.

2.3 TERMINAL USER INTERACTIONS

The interactions of the terminal user with the Form Executor are dependent on the functions of the keys on the terminal's keyboard. The keys on a terminal's keyboard are of two types. Any key that causes a character to be displayed on the terminal's screen is called a data key. The remaining keys, those that do not cause a character to be displayed, are called event keys. The event keys fall into two classes, the edit keys and the function keys. The difference lies in whether the application and/or the form designer has any control over the effect of the key.

TIFORM supports several types of terminals. For each terminal type, event keys are designated in different ways. For instance, on the 911 or 931 VDT, most event keys are designated by the use of keys that have a description of the event or a function number printed on the key top. For the 820 KSR, however, event keys are designated by the use of data keys in conjunction with the CTRL key.

In this paragraph, event keys are discussed in terms of the Form Executor response. The specific keys that are assigned to the operations are discussed in the paragraphs that describe the specific terminal types.

Several concepts are used repeatedly in these discussions that must be clear before proceeding. Recall that each Read command from the application can specify to read one or more fields. The Read command currently being executed by the Form Executor is called the *current application Read*. The Form Executor orders the fields that are specified by the current application Read. This ordering determines the next and previous fields relative to the current field, as well as the first and last fields of the current Read.

On an indexed Read, the first field read is the indexed field. The field being read by the Form Executor is called the *current field*. The cursor always resides within the current field. The terminal user can change any character of the current field or position the cursor anywhere within the current field by using the Next Character, Previous Character, Delete Character, Insert Character, or Repeat key without leaving the current field. Striking any other event key closes the current field.

When the current field is closed by a backward cursor movement, either Previous Field or Back Tab, or if an armed abort key is pressed, no editing or processing is performed on the closed field. When the current field is closed in any other way, the Form Executor immediately applies the field's editing and processing attributes to the field's new value. If any edit or process fails, a field error is declared.

A field error is processed as follows. Upon detecting an edit error, either at field edit time or at final validation time, the Form Executor displays a descriptive message. The exact manner of display is discussed in the paragraphs that describe each device type. It then issues a Read for a one-character field immediately following the message. The terminal user must close this field with the Return key or the Enter key, signifying that the error message has been seen. The Form Executor then makes the erroneous field the new current field, giving the user a chance to correct the error. Upon the successful completion of this Read, the error message is cleared from the screen.

If all the edits succeed, the field's new current value is accepted, and a decision is made regarding what field of the current application Read should be made the new current field.

The choice of a new current field depends on several things. If the event key that closed the field also closed the current Read, a new current field cannot be chosen until the next application Read is issued. If the closed field has a branching or conditional branching attribute, that attribute specifies the new current field. Otherwise, if a Forward Tab, Back Tab, Previous Field, Home, or Erase Input key was pressed, the new current field is chosen according to the particular key's rules as discussed in the paragraphs that describe each terminal device type. If none of these conditions are true, the new current field is the next field of the current application Read, starting at the just closed field. If the just closed field is the last field of the current application Read, the current application Read is closed as described in the following paragraphs.

If the current application Read is closed, either by the closing event key or because the last field of the Read was just closed, the Form Executor goes through several special steps. First, final validation is performed. All fields of the current application Read are reedited. If any field fails this validation edit, a field error is declared on that field and the current application Read is reopened. If all the validation editing succeeds, all the field processing attributes are put into effect. The values of the edited and processed fields and variables are then concatenated in the order of their membership in the group specified by the current application Read. This block of input data values is passed back to the application with the closing event key's code.

If the current application Read is closed by an armed abort key, no editing or final validation is performed. The block of input data values is built and sent to the application with the abort key's code and a form status of 03.

The current application Read can also be closed by exiting the Read's first field in a backward direction. In this case, no editing or final validation is performed. The block of input data values is built and sent to the application with a form status of 01.

Note that it is possible for the current application Read to be closed without an event key being pressed. If the last field of the current application Read has the autoskip attribute, that field closes automatically as soon as it is filled. An autoskip field close is treated like a forward field event key. Therefore, the forward field event key code (00) is returned when the last field is an autoskip field.

2.4 EDIT KEYS

The effects of the edit keys are fixed by the operating system and the Form Executor. The specific edit keys are discussed in the paragraphs that describe each terminal device type.

Several of the edit keys (Next Field, Return, and Skip) close the current application Read when pressed while in the Read's last field. The Enter key always closes the current application Read. If the terminal user presses any of these keys to close the current application Read, an event key code of 00 is returned. Also, a code of 00 is returned if the current application Read is closed by the last field being an autoskip field.

Two of the edit keys (Previous Field and Back Tab) close the current application Read when pressed while the cursor is in the Read's first field. If the terminal user closes the Read in this way, an event key code of 00 is returned together with a form status of 01.

2.5 USE OF FUNCTION KEYS IN A FORM

The function keys can be controlled completely by the form, or they can be controlled from the application program. Specific function keys are discussed in the paragraphs that describe each terminal device type.

You can associate a function key with a field in a form so that pressing the function key causes an immediate branch to the specified field. The association of a function key with a field is made in FDL by the FKEYS statement. Each FKEYS statement is local to the segment within which it is declared. Each segment can have its own FKEYS statement. The function keys specified in an FKEYS statement become active when that segment is prepared. If one of the specified function keys is pressed (and the application has not armed that function key itself), the current field is closed and the field associated with that key is made the current field if that field is within the current application Read. If the associated field is not within the current application Read, a warning beep is sounded, the current field is not closed, and the cursor is left where it is.

The application must implement all branching among segments. The following paragraph discusses intersegment branching.

2.6 USE OF FUNCTION KEYS BY AN APPLICATION

An application program uses the Arm Event Key and Disarm Event Key commands to specify which function keys the Form Executor recognizes.

When arming an event key, the application can specify whether the key is an abort key. If the terminal user presses an abort key, the Form Executor bypasses all edits of the current field, all final validation, and all field processing. The input data is returned to the application with the abort key's code as the event key code and a form status of 03. It is up to the application to process data returned after an abort key is pressed.

Assigning an abort key provides an escape mechanism that the terminal user can use to exit the current form immediately. For example, on the 931 VDT, the CMD key and F8 are common keys to arm as abort keys. When an armed nonabort event key is pressed, the Form Executor treats it like an Enter key, closing the current field and the current application Read, validating and processing all entered data, and returning the input data to the application. The only difference lies in the value returned as the event key code and the form status.

The application must implement all intersegment branching. The application must know what keys to arm and what segment to prepare for each key. To associate a function key with a segment, the application must arm the function key then execute the appropriate Prepare Segment command when it receives an event key code denoting that function key was pressed.

If a particular application wishes to implement a special Print Screen function other than the one supplied with TIFORM, it can arm the Print key. If the Print key is armed, the usual processing of this key by the Form Executor is overridden. It is treated like any other armed function key, and control is returned to the application. It is the responsibility of the application to perform the desired screen printing function and to keep the Print key armed across Prepare Segment commands.

While the capability is provided for the application to override the Form Executor's Print Screen function, it is strongly recommended that the Print key be maintained for some variation of screen printing. Arming the key for some other purpose is discouraged.

2.7 TERMINAL DEVICE TYPES SUPPORTED BY TIFORM

TIFORM supports the following device types:

- 911 VDT
- 915 VDT
- 931 VDT
- 940 EVT
- Business System terminal
- 820 KSR and other KSR types
- 810 printer, other printers, and sequential files

The device type is determined at run time by the Form Executor while processing an Open Form command received from the application. When the Executor opens the station specified by the Open Form command, the terminal type is determined, and the version of the Form Executor that supports that terminal type is bid. You can use a pathname for a terminal type. The pathname must be for a sequential file or for a nonexistent file.

For a description of how the DSR treats the ASCII characters for each key on the keyboards of the supported terminals, refer to either the *DX10 Operating System Application Programming Guide* or the *DNOS Supervisor Call (SVC) Reference Manual*, depending on the operating system you are using.

2.8 DISPLAY TERMINALS

The display terminals are the 911 VDT, 915 VDT, 931 VDT, 940 EVT, and the Business System terminal. When operating under TIFORM, these terminals are 24-line by 80-character VDTs with unique edit and function keys. Forms that include the FDL statement DEVICE = VDT – 2 match this screen size. However, other DEVICE types do not preclude the form from being executed in some manner. If no DEVICE type was used on an old form, the default is 24 lines and 80 characters per line. The mask of a segment and the location of all fields of a segment are displayed on the VDT screen as the result of a Prepare Segment command. The prior Open Form command does not clear the screen, and the screen is cleared by a Prepare Segment command only if the CLEAR = YES clause is included.

The following paragraphs discuss the display terminal edit keys, function keys, and error handling.

2.8.1 Display Terminal Edit Keys

Table 2-1 lists the function and name of each edit key. The key names are generic key names. To identify the specific key on the terminal you are using, refer to the table of key equivalents in Appendix A.

Table 2-1. Display Terminal Edit Key Functions and Names

TIFORM Function	Generic Name
Erase Field	Erase Field
Erase Input	Erase Input
Back Tab	Initialize Input
Print	Print
Up Arrow	Previous Line
Repeat	Repeat
Left Arrow	Previous Character
Home	Home
Right Arrow	Next Character
Insert Character	Insert Character
Down Arrow	Next Line
Delete Character	Delete Character
Close Read	Enter
Forward Character	Next Character
Backward Character	Previous Character
Forward Field	Next Field
Backward Field	Previous Field
Return	Return
Forward Tab	Forward Tab
Skip	Skip

The following paragraphs discuss the response when the terminal user presses each of the edit keys.

2.8.1.1 Erase Field Function — Erase Field Key. This key fills the current field with the form's fill character and positions the cursor at the beginning of the field. This function is performed by the operating system so its effect is almost instantaneous. This key does not close the current field.

2.8.1.2 Erase Input Function — Erase Input Key. This key fills all fields specified by the application's current Read with their default values and makes the first field of that Read the current field. The Form Executor displays filler values for those fields not having defaults. This key closes the current field when the user presses it. However, any data entered into the current field is discarded and the field's default value is installed.

2.8.1.3 Back Tab Function — Initialize Input Key. In response to this key, the Form Executor closes the current field and selects the next field by scanning backward through the fields of the current Read, looking for a tab stop field. If there is a tab stop field previous to the closed field, that field is selected. If there are no tab stop fields previous to the closed field, the key is treated like a Previous Field key.

2.8.1.4 Print Function — Print Key. The Print key is special. It is an armable event key, so the application can override any other meaning for it. Alternatively, if the application does not arm it, the Print key causes a Print Screen function to occur. The TIFORM Print Screen reads the current contents of the specified terminal's screen, and either copies it to the terminal's associated printer (if available) or places it in a file that it queues for later printing. See paragraph 2.11 for a detailed explanation of the Print Screen function.

The Print key closes the current field only when the application arms the key. If the key is being used to activate the Print Screen, it has no effect on the current field.

2.8.1.5 Up Arrow Function — Previous Line Key. This key positions the cursor in the first field directly above the current cursor position. If control mode 7 is off, the cursor is left in the same screen column in which it started. If control mode 7 is on, the cursor is moved to the first column of the selected field. If there is no field above the current cursor position, the cursor is not moved. This key cannot close the current application Read.

2.8.1.6 Repeat Function — Repeat or Typamatic Key. The Repeat key repeatedly transmits any other key pressed concurrently. As long as you hold the Repeat key down, the last pressed key is transmitted. This key is processed only by the 911 and 915 terminals. You can repeat any key on the keyboard in conjunction with the Repeat key. Its use in conjunction with a field-closing event key is not recommended.

The 931, 940, and Business System terminals do not have a Repeat key. Their keyboards have a typamatic feature. As long as you hold down any key on the keyboard, that key is repeated.

2.8.1.7 Left Arrow Function — Previous Character Key. This key moves the cursor one character position to the left within the current field. If the cursor is already in the leftmost character position of the current field, the cursor is left there and a warning beep is sounded. This key does not close the current field.

2.8.1.8 Home Function — Home Key. This key closes the current field, bypassing all edits and processing, and moves the cursor to the beginning of the first field of the current application Read.

2.8.1.9 Right Arrow Function — Next Character Key. This key moves the cursor one character position to the right within the current field. If the cursor is already in the rightmost character position of the current field, the cursor is left there and a warning beep is sounded. This key does not close the current field.

2.8.1.10 Insert Character Function — Insert Character Key. This key sets the input mode to *insert characters*, conditioning subsequent input keystrokes as follows. If you press a data key, the corresponding character is inserted in the current field at the current cursor position, moving all characters to the right of the cursor and the cursor itself one character position to the right. If a subsequent keystroke would cause characters to be lost off the right edge of the current field, a warning beep is sounded. The first subsequent nondata keystroke (a keystroke not entering any data in the field) returns the input mode to *noninsert*. If the character at the cursor position is a fill character and not a blank, this key gives a warning beep and does nothing. This key does not close the current field.

2.8.1.11 Down Arrow Function — Next Line Key. This key moves the cursor down on the screen. It places the cursor in the first field immediately below the current cursor position. If control mode 7 is off, the cursor is left in the same column of the screen in which it started. If control mode 7 is on, the cursor is moved to the first column of the selected field. If there is no field below the current cursor position, the cursor is not moved. This key cannot close the current application Read, but it always closes the current field regardless of whether the cursor is moved.

2.8.1.12 Delete Character Function — Delete Character Key. This key deletes the character at the cursor, moves all characters within the field to the right of the cursor one character position to the left, replaces the last character position of the field with the form's fill character, and leaves the cursor at its position prior to the keystroke. If there are no characters at or to the right of the cursor within the current field, this key gives a warning beep. This key does not close the current field.

2.8.1.13 Close Read Function — Enter Key. This key closes both the current field and the application Read, if possible, no matter when you press it. The final validation may reopen the Read due to edit errors. In particular, if any required fields are not yet entered, these are detected by final validation.

2.8.1.14 Forward Character Function — Next Character Key. Refer to paragraph 2.8.1.9.

2.8.1.15 Backward Character Function — Previous Character Key. Refer to paragraph 2.8.1.7.

2.8.1.16 Forward Field Function — Next Field Key. This key closes the current field and makes the next field in the current application Read the new current field. If the closed field is the last field of the current application Read, the cursor is left where it is, the current application Read is closed, and the input data is passed to the application with an event key code of 00.

2.8.1.17 Backward Field Function — Previous Field Key. This key closes the current field and makes the previous field in the current application Read the new current field without editing or processing the closed field. If the closed field is the first field of the current application Read, the application Read is closed in much the same way as if you pressed an armed abort key. No final validation is performed. A form status of 01 is returned to indicate the application Read was closed by moving backward out of the first field of the Read.

2.8.1.18 Return Function — Return Key. This key is identical to the Next Field key. It closes the current field and makes the next field in the current application Read the new current field. If the closed field is the last field of the current application Read, the cursor is left where it is, the current application Read is closed, and the input data is passed to the application with an event key code of 00.

2.8.1.19 Forward Tab Function — Forward Tab Key. This key closes the current field and selects the new current field by scanning forward through the fields of the current application Read looking for a field with the tab stop attribute. If there is a field with the tab stop attribute following the closed field, the first such field found is made the new current field. If there are no tab stop fields following the closed field, the key is treated like a Next Field key.

2.8.1.20 Skip Function — Skip Key. This key is similar to Return and Next Field. It always closes the current field. However, before doing so, it substitutes blanks for all characters in the field at and to the right of the cursor. The next field in the current application Read is made the new current field. If the closed field is the last field of the current application Read, the cursor is left where it is, the current application Read is closed, and the input data is passed to the application with an event key code of 00.

2.8.2 Display Terminal Function Keys

Where the functions and effects of the edit keys are determined by the operating system and the Form Executor, the effects of the function keys must be specified by the form designer and/or the application program. The function keys include F1 through F14, Print, and Command. (On the 931 VDT, you access F11 through F14 with (SHIFT) F1 through (SHIFT) F14. Refer to Appendix A for a complete list of function key mapping.) The function keys may be specified by the form designer for field branching, or they may be armed and interpreted by the application program.

If a function key is given multiple functions, application arming takes precedence over field branching. If the application arms a function key, that arming takes precedence over any other use of that key.

If a function key is given no function, its use by the terminal operator causes an error. If an unassigned function key is pressed, the Form Executor responds with a warning beep. Other than the beep, the use of an unassigned function key is ignored.

Table 2-2 indicates the names to be used by a form designer and the codes to be used in an application program to arm and disarm the various function keys. The application receives these codes as the event key code on a Read command completion if one of these keys terminated the Read. The Print key is included because an application can arm it. It is not classified as a function key, and you cannot use it in a form.

Table 2-2. Display Terminal Function Key Names and Codes

Generic Key Name	Application Name	Code
F1	01	01
F2	02	02
F3	03	03
F4	04	04
F5	05	05
F6	06	06
F7	07	07
F8	08	08
F9	09	09
F10	10	10
F11	11	11
F12	12	12
F13	13	13
F14	14	14
Command	40	40
Print	—	49

2.8.3 Display Terminal Error Handling

A field error can occur for many reasons. However, the handling of errors remains the same regardless of the cause. If an error occurs, an error message is displayed on the bottom line of the screen and the cursor is positioned at the right of that line. The terminal user must acknowledge the error by pressing the Enter key or the Return key. The cursor is then positioned in the field in error and on the character in error if the error was due to an invalid character. The error message remains on the screen until the user corrects the error and closes the field.

2.9 820 KSR AND OTHER KSR TYPES

The discussion in this paragraph uses the 820 KSR as the representative of the KSR type of terminals. However, you can use other KSR types.

The generic key names on the chart in Appendix A are valid for KSR terminals. However, since this paragraph is focusing on the 820, the key names for the 820 are used instead of generic key names.

The current image of the KSR screen is maintained by the Form Executor, and, during normal operations, only parts of the screen are printed at a time. Forms that include the FDL statement DEVICE = KSR – 1 (66 lines by 80 characters) or DEVICE = KSR – 2 (66 lines by 132 characters) are appropriate. However, other DEVICE types are not precluded from execution on the 820 KSR. Old forms that did not include the DEVICE statement are defaulted to 24 lines by 80 characters.

The event keys on an 820 are implemented by the use of the control (CTRL) key in conjunction with certain data keys. Table 2-3 and Table 2-4 list the key pairs that implement the event key functions on the 820. The Form Executor supports several modes of operation that are unique to the KSR-type terminals.

The Form Executor supports several modes of operation that are unique to the KSR-type terminals: formatted input, unformatted input, delayed write mode, and immediate write mode. The following paragraphs discuss these modes of operation. They also discuss error handling, field mask handling, edit keys, function keys, and the Print Screen key.

2.9.1 Formatted Versus Unformatted Input

An 820 KSR can operate in the formatted or unformatted input mode. In the formatted input mode, the entire KSR screen is available for printing. In the unformatted input mode, only the field prompts are printed along with the current contents of fields into which data is to be entered by the user. The Print key (CTRL /Y) always prints the segment mask and the current contents of all fields of the segment on the 820. You can use the Print key in either mode.

2.9.1.1 Formatted Input. The normal mode of operation for the terminal is the formatted mode. A KSR in the formatted mode works in the following manner. After an Open Form command starts the Form Executor and the desired form is selected, a Prepare Segment command from the application program identifies the segment to use. A page eject is issued only if the segment to prepare includes a ,CLEAR = YES clause on the segment mask statement.

Assume a Read command is issued, and the first field of the group is in line three. Lines one through three of the screen are printed. These lines contain the entire contents of the screen, both background mask information, and the locations of all fields (marked by underscores or the current contents of the fields) that are in the first three lines. After the third line is printed, a line feed is issued, and the printhead is positioned under the first character position of the field in the third line that is the first field included in the Read.

After input for that field is complete, the printhead is moved under the next field in the line that is included in the Read. Although the contents of fields on the third line that are not in the group being read are printed, the printhead is not positioned under them. When all fields of the group that are located on line three are read, all lines from line three down to, and including, the line that contains other fields included in the Read are printed. Another line feed is issued, and the printhead is positioned under the first field in the line that is included in the group.

This process is continued until all fields included in the group are read. At this time, the data read is returned to the application.

If a Read for the same group is issued again, a line feed is issued by the Form Executor, line three is printed again, and the printhead is once again positioned under line three. Lines one and two are not printed because the printhead was on a line of the virtual screen that was below the line that contained the first field to read. Lines one and two were printed earlier. After input for all fields of line three is complete, the Read proceeds through the group as described previously.

In this mode, fields are printed in the proper position relative to the left edge of the screen. Prompts specified by the PROMPT statement in the FDL of the segment are not displayed.

2.9.1.2 Unformatted Input. In the unformatted input mode, the background mask is not printed. A prompt for the first field of the group to read is printed, if one was specified in the FDL, followed immediately by the current contents of the field. A line feed is issued, and the cursor is positioned beneath the first character of the field. When the field is complete, the same series of operations is repeated for the next field. The series is repeated until all fields are complete and the Read is closed.

In this mode, the positions of the fields of the group being read, or of fields of other groups, are not relevant other than to determine the order of the fields of the group.

The normal mode of operation of the Form Executor is the formatted input mode. To switch to the unformatted input mode, the application program must send a Control Functions command with a condition code of +5. A condition code of −5 returns the Executor to the formatted input mode. (See paragraph 5.10.)

2.9.2 Delayed Versus Immediate Write Mode

While the KSR is in the delayed write mode, information passed to the terminal by the use of a Write command from the application is stored in the virtual screen maintained by the Form Executor, but is not printed. Information written to the fields of the group is printed only as a result of a subsequent Read command or some user action, such as pressing the Print key.

When a terminal is in the delayed write mode, the contents of a field are printed following these events:

- Prior to a prompt for new input for the field if a Read is issued
- When the Print key (CTRL /Y) is pressed
- When a new segment is prepared or the form is closed and the current contents of the field are not yet printed

In the immediate write mode, the contents of the fields of the group are printed immediately following the receipt of the contents by the Executor. If the terminal is in the formatted input mode, the contents of the lines containing the fields of the written group are printed. This includes background mask and contents of fields of the written group, and of other fields on the same lines.

If the terminal is in the unformatted input mode, the prompts and values of the fields of the group referenced in the Write command are printed and left-justified, without regard for column position. Background mask information is not printed.

If a terminal is in the immediate write mode, the contents of a field are printed following these events:

- When a field is assigned its default value during the preparation of a segment
- When a Write command is issued by the application
- When a field's postentry attributes are processed immediately after that field is read
- When a field's postentry attributes are processed a second time for final validation

- When the application issues a Write With Reply command
- When the application issues a Reset command and the contents of a field are restored to its default value
- When a field receives a new value as a result of another field's COPY TO attribute being processed
- When the Print key is pressed
- When a new segment is prepared, or the form is closed, if the contents of the field are not yet printed

The terminal is normally in the delayed write mode. In order to switch the terminal to the immediate write mode, the application must send a Control Functions command with a condition code of +4. The terminal can be switched back to the delayed write mode by sending a condition code of −4.

2.9.3 820 Error Handling

When a field error occurs on an 820 KSR terminal, two line feeds are issued and an error message is printed. Then either the entire line containing the error (formatted input mode) is printed, or the prompt for the field in error and the contents of the field (unformatted input mode) are printed. Another line feed is issued, and the printhead is positioned below (formatted input mode) or next to (unformatted input mode) the field in error. The user can then correct the error. This process is repeated until the error is corrected.

2.9.4 820 Field Mask Handling

Field masks are not normally printed on the 820. However, the user can request the printing of the field mask for a given field at any time. To print a field mask, the user presses CTRL /−. In response, two line feeds are issued and the field mask, if one exists for that field, is printed. Two more line feeds are issued, and the line containing the current field (formatted input mode) or the prompt and the contents of the current field (unformatted input mode) are printed. The printhead is then positioned under (formatted input mode) or next to (unformatted input mode) the current field.

If no field mask exists for the current field, the line containing the current field, or the field and prompt, are reprinted.

If the application program arms CRTL /−, the application program's use of the key takes precedence and any field mask is not printed.

2.9.5 820 Edit Keys

Table 2-3 lists the edit functions and the key combinations you use to invoke the functions.

Table 2-3. KSR Edit Key Functions and Names

TIFORM Function	Silent 700™ and 820 Key Names
Erase Field	DEL
Erase Input	CTRL /N
Back Tab	CTRL /O
Print	CTRL /Y
Up Arrow	CTRL /U
Repeat	(data key held down)
Left Arrow	CTRL /H
Home	CTRL /L
Right Arrow	(not available)
Insert Character	(not available)
Down Arrow	CTRL /J
Delete Character	(not available)
Forward Character	(not available)
Backward Character	CTRL /H
Forward Field	CTRL /M
Backward Field	CTRL /T
Return	RETURN and CTRL/M
Forward Tab	CTRL /I
Skip	CTRL /K
Close Read	CTRL /S

The following paragraphs discuss the response when the terminal user selects each of the edit functions.

2.9.5.1 Erase Field Function — DEL. This key deletes all characters that are entered for the current field. The current Read is not closed. The printhead moves to the next line and you must press RETURN to close the field.

2.9.5.2 Erase Input Function — CTRL/N. This key closes the current field and assigns each field specified by the application's current Read its default value. You are prompted for the first field of the Read. On an 820, each field's default is printed only in the immediate write mode.

2.9.5.3 Back Tab Function — CTRL/O. This key closes the current field. The Form Executor scans backward through the fields of the current application Read looking for a tab stop field. If a tab stop field is found, you are prompted for that field. Otherwise, the key is treated like a Backward Field key.

2.9.5.4 Print Function — CTRL/Y. Refer to paragraph 2.9.7.

Silent 700 is a trademark of Texas Instruments Incorporated.

2.9.5.5 Up Arrow Function — CTRL/U. This key prompts for the first field above the current field (based on the starting column position). If no such field exists, the current field is reprompted.

2.9.5.6 Repeat Function — Typamatic. You select the repeat function by holding down the key to be repeated for as long as necessary.

2.9.5.7 Left Arrow Function — CTRL/H. This key positions the printhead on the next line and moves it one position to the left. Any data entered replaces the data on the line directly above it. If you press this key more than once, the printhead is positioned one character to the left for each time you press the key, but only one line feed is sent to the terminal.

2.9.5.8 Home Function — CTRL/L. This key closes the current field, bypassing all edits and processing. You are prompted for the first field of the current application Read.

2.9.5.9 Down Arrow Function — CTRL/J. This key prompts for the first field below the current field (based upon the starting column position). If no such field exists, the prompt reappears for the current field.

2.9.5.10 Backward Character Function — CTRL/H. Refer to paragraph 2.9.5.7.

2.9.5.11 Forward Field Function — CTRL/M. This key closes the current field. If there are more fields to read in the current application Read, you are prompted for the next field.

2.9.5.12 Backward Field Function — CTRL/T. This key closes the current field without editing or processing it. If the current field is not the first field in the Read, you are prompted for the previous field. If the closed field is the first field, the current application Read is closed in the same way as if you pressed an armed abort key. If you have not entered any input at the time you press this key, the original contents of the field are not changed.

2.9.5.13 Return Function — RETURN and CTRL/M. Refer to paragraph 2.9.5.11.

2.9.5.14 Forward Tab Function — CTRL/I. This key closes the current field and selects the next field by scanning forward through the fields of the current application Read looking for a tab stop field. If a tab stop field is found, you are prompted for it. If none is found, the key is treated like a RETURN key.

2.9.5.15 Skip Function — CTRL/K. This key deletes all characters entered for the current field and closes the current field. If there are more fields to read in the current application Read, you are prompted for the next field.

2.9.5.16 Close Read Function — CTRL/S. This key closes both the current field and the application Read, if possible, no matter when you press it. You do not see the values for any fields that are not yet read unless the immediate write mode is enabled by the application. Should any field fail final validation, an error message is printed and you are prompted for a new value.

2.9.5.17 Right Arrow, Forward Character, Insert Character, and Delete Character. These functions are not available on an 820 terminal.

2.9.6 820 Function Keys

Where the functions and effects of the edit keys are determined by the operating system and the Form Executor, the effects of the function keys must be specified by the form designer and/or the application program. These keys, which are implemented by the simultaneous use of the CTRL key and a data key, can be used by the form designer for field branching, or they can be armed and interpreted by the application program.

If a function key is given multiple functions, application arming takes precedence over field branching. If the application arms a function key, that arming takes precedence over any other use of that key.

If a function key is given no function, its use by the terminal operator causes an error. If an unassigned function key is pressed, the Form Executor responds with a warning beep. Other than the beep, the use of an unassigned function key is ignored.

Table 2-4 indicates the names to be used by a form designer and the codes to be used in an application program to arm and disarm the various function keys. The application receives these codes as the event key code on a Read command completion if one of these keys terminated the Read. The Print key is included because an application can arm it. It is not classified as a function key, and you cannot use it in a form. The equivalent VDT function key is also given (in parentheses) for convenience.

Table 2-4. KSR Function Key Names and Codes

Silent 700 Keyboard	820 Keyboard	Application Name	Code
CTRL /A	CTRL /A (F1)	01	01
CTRL /B	CTRL /B (F2)	02	02
CTRL /C	CTRL /C (F3)	03	03
CTRL /D	CTRL /D (F4)	04	04
CTRL /E	CTRL /E (F5)	05	05
CTRL /F	CTRL /F (F6)	06	06
CTRL /V	CTRL /V (F7)	07	07
CTRL /W	CTRL /W (F8)	08	08
CTRL /3	CTRL /[(F9)	09	09
CTRL /Z	CTRL /Z (F10)	10	10
CTRL /\	CTRL /\ (F11)	11	11
CTRL /]	CTRL /{ (F12)	12	12
CTRL /	CTRL / = (F13)	13	13
CTRL /_	CTRL / – (F14)	14	14
CTRL /X	CTRL /X (Command)	40	40
CTRL /Y	CTRL /Y (Print)	—	49

2.9.7 820 Print Screen

When the terminal user presses the Print Screen key (CTRL /Y), a page eject is issued and the contents of the virtual screen are printed. This screen contains the background for the currently prepared segment and the current contents of all of the fields. After the screen is printed, another page eject is issued, and the line containing the current field, or the current field and its prompt, are printed. The printhead is positioned under (formatted input mode) or next to (unformatted input mode) the current field, and you can continue to enter data.

2.10 810 PRINTER, OTHER PRINTERS, AND SEQUENTIAL FILES

The terminal type can be a printer or a sequential file. If the specified pathname is that of a nonexistent file, a sequential file is created provided that the directory containing this file exists and has an available entry. Relative record files and file types other than sequential are not supported, and a system file error code is returned following the Open Form command.

Using a printer or a sequential file as a terminal type allows the application program to record entire screens of information. The application does this when it leaves the terminal in the delayed write mode, prepares the desired segment, then issues Write commands. Nothing is written to the printer or the file as a result of these operations. The application then issues a Print Screen command to cause the segment mask and the contents of all fields of the segment to be written to the terminal as a series of sequential records.

You can place a file or printer in the immediate write mode. If you do this, the contents of Write commands are printed or written to the file as if to a terminal. This is not recommended.

An attempt to read from a printer results in a system error being returned to the application. If the application attempts to read from a sequential file, a system error may not result, but the results are unpredictable.

2.11 TIFORM PRINT KEY

When you press the Print key while executing a form, the Form Executor performs all the work necessary for printing the contents of the terminal's screen. The Form Executor then displays the message to let you know the screen is printed. You must acknowledge the message by pressing the Return key. Following acknowledgement, the Form Executor awaits the next keyboard input.

A terminal can have a private print directory associated with it in the terminal's .S$TIFORM.PRINT.TERMINAL record. If so, the result of a Print Screen request for that terminal is always placed in the specified directory. It is never printed directly. The following discussion uses the public print directory's name, .S$TIFORM.PRINT, whenever a print directory name is needed. If the terminal has a private print directory, that directory's name is used in place of .S$TIFORM.PRINT. The file .S$TIFORM.PRINT.TERMINAL is the one exception to this substitution. This file is unique and always resides in the .S$TIFORM.PRINT directory.

2.11.1 Print Key Function's Execution

When the Print key is activated, it immediately reads the contents of the activating terminal's screen and stores it in memory. It then looks in the file .S$TIFORM.PRINT.TERMINAL for the terminal/printer association. If the activating terminal's name is not in this file or the file does not exist, the default printer name LP01 is used.

Having selected the printer to use, the Print key tries to acquire it exclusively. If the printer is free, this operation is successful. If the printer is busy, the screen is stored in a file. Depending on the success or failure of this operation, the task composes one of the following messages:

```
SCREEN HAS BEEN SENT TO PRINTER <device name>
SCREEN HAS BEEN SENT TO A FILE .S$$TIFORM.PRINT.AAAA
```

The composed message is then sent to the Form Executor for display and the Executor is activated, thereby minimizing the delay that you experience.

If the terminal's record in .S$TIFORM.PRINT.TERMINAL specified a private print directory name, no attempt is made to acquire a printer. The ON QUEUE message is sent to the terminal, and the Print key continues as if it had been unable to acquire the terminal's printer.

The actions of the Print key now depend on whether the printer is acquired. If it is acquired, the memory image of the terminal's screen is sent directly to the printer. All graphics characters are translated as described in Appendix E. The Print key then terminates execution, having printed the screen.

If the printer is not acquired, it copies its memory image of the terminal's screen to a file named .S$TIFORM.PRINT.< unique name>, again translating all graphics characters. A Print File (PF) command with DEL = Y is then added to the file .S$TIFORM.PRINT.QUEUE for the file just created. The Print key then terminates execution, having queued the screen image for later printing and deletion. To print any queued files, you must execute the Execute Batch (XB) SCI command as follows:

```
XB  INPUT=.S$TIFORM.PRINT.QUEUE, LIST=DUMY
```

2.11.2 Print Key Files

The Print key assumes that the directory .S$TIFORM.PRINT is available for its private use. Within this directory, it looks for certain files and constructs certain other files, as described in the following paragraphs.

2.11.2.1 Terminal File. The association of terminals with printers and private print directories is controlled by the contents of the following:

.S$TIFORM.PRINT.TERMINAL

For each terminal for which you want a printer other than the default printer (LP01), a record must exist in this file formatted as follows:

< terminal name> < printer name> < private print directory>

You can list the terminals in any order with any number of blanks on either side of the names in each line. The Print key ignores any invalid records in the file. If you activate the Print key for a terminal for which no valid record exists, or if the file .S$TIFORM.PRINT.TERMINAL does not exist, the default printer LP01 is used.

The third field on a .S$TIFORM.PRINT.TERMINAL record is optional. If present, it specifies the pathname of a directory to use as the terminal's private print directory. If you specify a private print directory, the Print key always queues the terminal's printed screens in that directory. Other than this forced queueing, a private print directory is used exactly like .S$TIFORM.PRINT is used. It has its own .QUEUE and .FLAG files, and the names of its files containing screen images are .AAAA through .ZZZZ.

The following terminal file demonstrates the options available. At ST01, the output always goes to the TIFORM queue file. At ST02, the output goes to LP02, if available, or else to the queue file. At ST03, the output always goes to the private directory .TEMP.PRIVATE. At ST04, the output always goes to the private directory DS02.PRINTKEY. (Including LP01 has no effect.) At all other stations, the output goes to LP01, if available, or else to the TIFORM queue file.

```
ST01 DUMY
ST02 LP02
ST03 DUMY .TEMP.PRIVATE
ST04 LP01 DS02.PRINTKEY
```

2.11.2.2 Queue File. If the contents of the screen are saved in a file, a batch stream to print this file is placed in the following file:

.S$TIFORM.PRINT.QUEUE

The file .S$TIFORM.PRINT.QUEUE contains commands to print all files that the Print Key creates since the last time the .S$TIFORM.PRINT.QUEUE batch stream was executed. The first record in the queue file is a comment line containing the most recently generated file name. This line is followed by the Print File (PF) commands. The last line in the queue file is a command to delete the flag file, as described in the next paragraph.

If the Print key is activated twice while the printer is busy, .S$TIFORM.PRINT.QUEUE looks as follows:

```
* AAAB
PF FILE=.S$TIFORM.PRINT.AAAA,LIST=LPnn,DEL=Y
PF FILE=.S$TIFORM.PRINT.AAAB,LIST=LPnn,DEL=Y
DF PATHNAME=.S$TIFORM.PRINT.FLAG
```

In order to print screens on this queue, you must issue an XB command for the file .S$TIFORM.PRINT.QUEUE from a terminal at which SCI is active.

2.11.2.3 Flag File. The file .S$TIFORM.PRINT.FLAG is used to tell the Print Key function whether the batch stream in .S$TIFORM.PRINT.QUEUE has executed. If the flag file exists, the batch stream has not executed and a new file that needs to be printed is put on the end of the queue. If the flag file does not exist, the batch stream has executed and the queue file is rebuilt with the only file on the queue being the new file that needs to be printed. In this case, the flag file is recreated. The file .S$TIFORM.PRINT.FLAG is never written to or read; it is only referenced to determine whether it exists.

3

Form Definition Language

3.1 INTRODUCTION

The form definition language (FDL) is a structured nonprocedural language for defining forms. The FDL compiler translates the FDL statements into a series of overlays in a program file that the TIFORM Form Executor can execute directly. You can use either of two methods to build the FDL for a form:

- Describe the form directly in FDL.
- Use the Interactive Screen Generator/Editor (ISGE).

Although you can create a form directly in FDL, it is usually more efficient to use ISGE for this purpose. However, when you modify a compiled form, it is more efficient to enter changes directly into the source file in FDL rather than to reenter ISGE. When you use FDL statements to design a form, you must enter those statements into a source file using the Text Editor. (Refer to either the *DX10 Operating System Text Editor Manual* or the *DNOS Text Editor Reference Manual*.)

This section describes the overall structure of the FDL language, provides a sample form definition, defines FDL syntax, and describes the FDL compiler. It also presents a functional description of the FDL statements and then describes each FDL statement in alphabetical order, beginning with paragraph 3.2.

3.1.1 Overall Structure of the Language

FDL is a block-structured language. Each block defines and names a component of a form. The block structure also establishes contexts for FDL statements within the definition. Appendix F provides a quick reference to the valid FDL statements in each context.

FDL contains seven types of blocks forming a three-level hierarchy. The outermost level is the form block, which contains segment blocks and segment mask blocks. A segment block contains field blocks, edit set blocks, condition blocks, and field mask blocks. Figure 3-1 shows the general relationships of these blocks and the major FDL statement types.

Each block starts with a statement that defines the type and name of the block and ends with an optional END statement. These statements have the following formats:

```
type block. comments
END type block.
```

The END statement signals the end of the definition of the specified block. If you omit this optional statement, the FDL compiler treats the next block definition as if it were a combination end block/begin block statement. The FDL statements after each starting block statement define the structure and attributes of the block.

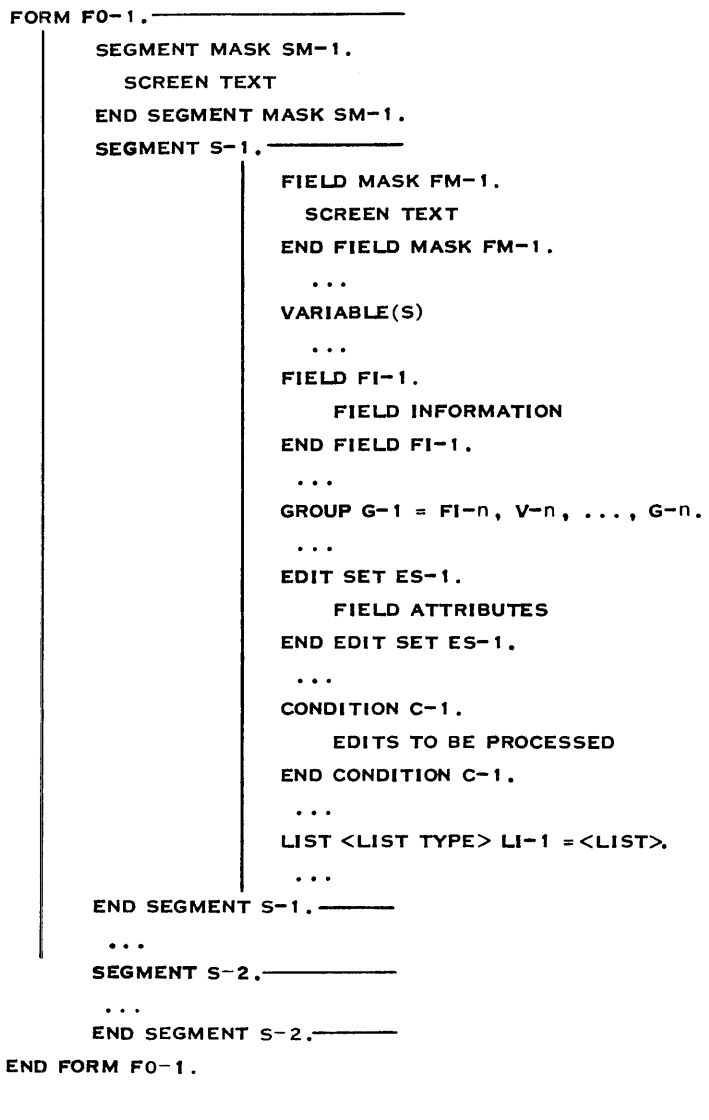


Figure 3-1. Relationships Among FDL Block Types

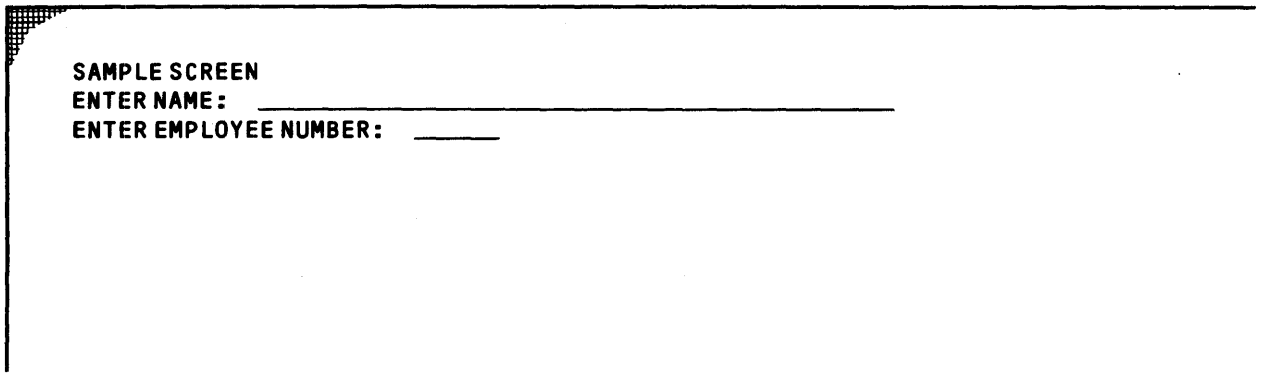
3.1.2 Sample Form Definition

The following example shows a form written in FDL without the optional END statements. You may find it useful to refer to this example when reading the definitions of FDL statements. This example defines a three-line segment mask and two fields.

EXAMPLE

```
FORM SAMPLE,999990,00.        Name, part number, revision number
ENABLE CONTROL MODE 6.        Turn control function 6 on.
SEGMENT MASK M1,CLEAR=Y.      Name, clear before mask display.
  M(1,2) 'SAMPLE SCREEN'.              Mask element for header.
  M(2,1) 'ENTER NAME:'.                Mask element for NAME field.
  M(3,1) 'ENTER EMPLOYEE NUMBER:'.     Mask element for NUMBER field.
.
SEGMENT SEG1,(SAMPLE),M1.  Insert SEG1 in form SAMPLE.
.
.                     NAME
  FIELD NAME.
    POS (2,15)L20.   Field begins on row 2, column 15, with a length of 20
    CHAR LIST=ALPHA. Accept characters defined in List Char Alpha
    JUST L,FILL=' '. Left justify and fill to the right with blanks
.
.                     NUMBER
  FIELD NUMBER.
    POS (3,25)L6.    Field begins on row 3, column 25, with a length of 6
    CHAR LIST=NUM.   Accept characters defined in List Char Num
    JUST R,FILL='0'. Right justify and fill to the left with 0's
.
  GROUP G1 = NAME, NUMBER. Group defined to be preceding two fields
.                          in order of position
.
  LIST CHARACTER ALPHA=A..Z,BLANK. Only A thru Z and blank are accepted.
.
  LIST CHARACTER NUM=0..9,'.','-'.  Only 0 thru 9, decimal, and dash are
.                                   accepted.
.
```

This example does not use all of the FDL statements, but it does create an executable form when you compile it. If you compile the example and then use the Form Tester utility to open the form SAMPLE and prepare the segment SEG1, the segment appears as follows:



The definition of each FDL statement is accompanied by one or more examples throughout this section.

3.1.3 Syntax Notation

The following conventions are used to describe the syntax of FDL:

Notation	Meaning
Uppercase	Keywords that must be entered as shown. You can omit the rightmost letters shown in italics.
Lowercase	Generic terms that represent the names, literals, and numbers used in your application: • A name consists of a letter followed by up to five additional letters, numerals, dashes, and dollar signs. • A literal consists of up to 78 characters enclosed by single quotes. You can use two single quotes to represent a single quote within the text. • A number is a signed integer, unless stated otherwise.
Braces	Enclose lists of items from which you choose one. Vertical bars separate items on the list.
Brackets	Enclose lists of items from which you choose one or none. Vertical bars separate items on the list.
Three dots	Mean you can enter the preceding item more than once, using commas as separators.
Punctuation	Must be entered as shown (other than the three dots): = , . " * ;
Comments	Can be included in the FDL source in four ways: • Each FDL statement ends with a period followed by a space. Any text to the right of the period is a comment. • Any text to the right of an exclamation point is a comment (unless the exclamation point is part of a literal). • Any text on a line beginning with a period is a comment. • Any text on a line beginning with a slash is a comment to be printed on a new page.

3.1.4 Executing the FDL Compiler

You can execute the FDL compiler (FDLC) in either of two ways. One way is to choose the compile segment option when terminating a session with the ISGE. Section 4 describes this process. The second way to execute the FDLC is to use the XFDLC command as described here.

When you enter the XFDLC command, you receive the following prompts:

```
          SOURCE FORM PATHNAME: acnm
OBJECT PROGRAM FILE PATHNAME: acnm
        LISTING FILE PATHNAME: acnm
         ERRORS FILE PATHNAME: acnm
                      OPTIONS: options
```

SOURCE FORM PATHNAME

Enter the pathname of the input file containing FDL statements that describe the form, segment(s), or segment mask(s) to compile. This parameter is required.

OBJECT PROGRAM FILE PATHNAME

Enter the pathname of the output file to which the executable code is written. This file must be an existing program file. The FDLC cannot create a program file. The object segments, segment masks, and form root are stored in this file as relocatable overlays. The program file must have enough available overlay entries for the form root and each segment mask and segment declared in your FDL source. The FDLC places each form root, segment mask, and segment into a different overlay with the same name used in your FDL block declarations. Each form root, segment mask, and segment must have a unique name within the program file. If you compile the example in paragraph 3.1.2, there are three overlays. This parameter is required.

LISTING FILE PATHNAME

Enter the pathname of the file to which the compiled listing is written. This parameter is required. The listing file contains the following:

- Access names table containing the pathnames of all files specified in the XFDLC command
- Image of each source record from the source file
- Compiler-produced diagnostics
- FDLC termination message

ERRORS FILE PATHNAME

Enter the pathname of the output file to which compilation errors are written. It contains all erroneous FDL source statements and the compiler-generated diagnostics consequently produced. This parameter is not required. It defaults to the terminal local file (TLF) if you do not specify a pathname.

OPTIONS

Enter the special compiler functions that you want enabled. Use only one of the following options at a time. This parameter is not required.

NOSYMT
: Disables generation of a symbol table in all segments being compiled. The FDLC usually builds a full symbol table in each object segment. Each name table entry is eight bytes long, and there is an entry for each field, field array member, variable, group, and list defined in the segment. The size of object segments can be reduced through the use of this option. This option is recommended if memory conservation is required. The TIFORM Release Information discusses memory usage.

DELSEG
: Deletes all device-dependent segments and segment masks that have the same name as the segment for which you specify this option. The segment for which you specify this option is not deleted.

The XFDLC procedure executes as a background task, and displays a termination message on the terminal after the FDL compilation is complete. Appendix C provides a description of each termination message. If the termination message indicates an error-free compilation, the form is ready for execution. You can use the Form Tester utility to verify that your FDL source defines the desired form. However, if any errors occur, you can view the errors file by entering the Show File (SF) command.

NOTE

Always correct all errors and recompile before attempting to execute a form. The results of executing an erroneous form, segment, or segment mask are unpredictable.

3.1.5 Functional Description of FDL Statements

The following paragraphs describe the FDL statements used in the seven block types.

3.1.5.1 Form Block. The form block is the basic unit by which a form is identified. The segments and segment masks of the form are defined in the form block. The following statements are used in form blocks:

- DEVICE
- DISABLE CONTROL MODE
- ENABLE CONTROL MODE
- END FORM
- FILLER
- FORM

The form block starts with a FORM statement and can end with an optional END FORM statement. When you compile a form block, a form root that contains the names of only the segments and segment masks within the block is stored in the program file. You must store the segments and segment masks of a form in the same program file as the form's root.

Note that you can compile segments separately from a form block, and you do not need to compile all the segments of a form at once. Each separate segment compilation updates the form's root overlay in the program file. A compilation of a form block, however, replaces the form root overlay. After the compilation of a form block, the form root contains the names of only those segments defined within the form block.

3.1.5.2 Segment Block. A segment block contains the entire description of a segment's variables, groups, fields, and their attributes. The following statements are used in a segment block:

- DEVICE
- END SEGMENT
- ERROR MESSAGE
- EXTERNAL
- FILLER
- FKEYS
- GROUP
- LIST CHARACTER
- LIST LENGTH
- LIST RANGE
- LIST SUBSTITUTE
- LIST TABLE
- MASK
- ORDERED GROUP
- SEGMENT
- VARIABLE

The segment block starts with a SEGMENT statement and ends with an optional END SEGMENT statement. You can compile a segment block apart from its form root.

In order to compile a segment apart from a segment mask, the SEGMENT statement must not specify a segment mask. If the SEGMENT statement does specify a segment mask, the segment mask block must immediately precede the segment block, and you must compile the segment mask block and the segment block together. You can compile only one segment mask and segment at a time. If you compile a segment without a segment mask, no mask statements (see paragraph 3.31) can exist within the segment block.

EXAMPLE

```
SEGMENT MASK APPLE,CLEAR=Y.     You must compile the SEGMENT MASK
  Screen text                   APPLE with the SEGMENT PIE in
                                the order they are shown here.

SEGMENT PIE,(FORM01),APPLE.     See paragraph 3.1.5.3 for segment
  Field information             mask blocks.
```

3.1.5.3 Background Mask Blocks. A mask block defines constant background text that the application displays but does not change. The two kinds of mask blocks are segment mask blocks and field mask blocks. The following statements are used in mask blocks:

- DISPLAY
- END FIELD MASK
- END SEGMENT MASK
- FIELD MASK
- MASK
- SEGMENT MASK

A segment mask block defines background text to be stored separate from any segment. Each segment mask is stored as a separate overlay in a program file. Any segment in that program file can associate itself with any segment mask in that program file, causing the segment mask to be displayed whenever the segment is activated by a Prepare Segment command from an application program. You cannot compile a segment mask block without an associated segment block. A simple segment mask follows.

EXAMPLE

```
SEGMENT MASK SEGMSK,CLEAR=NO.  Leave screen as is before display.
  M(1,20) 'Name:'.
  DISPLAY GR=Y.                Turn on graphics.
  M(10,1) 'LLLLLLLLLL'.        Write horizontal graphic line.
  DISPLAY GR=N, BR=Y.          Turn off graphics; make bright.
  M(15,1) 'Address:'.

SEGMENT SEGM,FORM01,SEGMSK.    An empty segment.
```

A field mask block is defined within a segment. Whenever the segment containing the field mask is activated, the field mask is loaded into memory with the segment. The field mask is not displayed until a field specifying its name in a DISPLAY MASK statement becomes the current field. The named field mask is displayed prior to reading that field. You can clear the field mask from the screen when the field is exited by using the POSTCLEAR attribute. A simple field mask follows.

EXAMPLE

```
FIELD SPECIAL.                              Display the field mask when the
  POS (23,1)L10.                            field is entered, and clear the
  DISPLAY MASK FLDMSK,POSTCLEAR.            mask when the input is valid.
.
FIELD MASK FLDMSK,CLEAR=N.                  Do not clear screen.
  M(23,12) 'This is for a special case.'. Displayed text.
```

The actual text of a mask is defined by background text statements (see paragraph 3.31). Background text statements are defined in the segment mask block or the segment block.

3.1.5.4 Field Block. The field is the basic unit of I/O data within a segment. Each field is specified by a field block. A field block starts with a FIELD statement specifying the field's name. The block can end with an optional END FIELD statement. The field block contains field attribute statements that specify the location, the size, and the edit/processing attributes for that field. The statements that are allowed within a field block are as follows:

- ARRAY
- AUTOSKIP
- BRANCH
- CHARACTER LIST
- COPY
- DEFAULT
- DISPLAY
- DISPLAY MASK
- END FIELD
- EXTERNAL
- FIELD
- FILLER

- GRAPHICS INPUT
- IF
- JUSTIFY
- LENGTH LIST
- MINIMUM LENGTH
- NO ENTRY
- NOAUTOSKIP
- NOTAB
- NOTREQUIRED
- NOVALIDATE
- NUMERIC
- OUTPUT
- PASS/FAIL
- POSITION
- PROMPT
- RANGE LIST
- REQUIRED
- SAME AS
- SCALE
- SUBSTITUTE LIST
- TAB
- TABLE LIST
- TERMINATE READ
- VALUE

3.1.5.5 Condition Block. A condition block lists the criteria used by an IF statement in selecting one of its options: THEN or ELSE. A condition block starts with a CONDITION statement and can end with an optional END CONDITION statement. The condition block can contain one or more of the following edit statements:

- CHARACTER LIST
- CONDITION
- END CONDITION
- IF
- LENGTH LIST
- MINIMUM LENGTH
- PASS/FAIL
- RANGE LIST
- TABLE LIST

The following condition block might be used to check for a valid year of manufacture, where the user can enter the year as two digits or four. The CHARACTER LIST statement makes sure the user enters only digits. The LENGTH LIST statement rules out entries with other than two or four digits. The RANGE LIST statement checks for years after 1984.

```
CONDITION YEAR.
CHARACTER LIST = DIGITS.
LENGTH LIST = YLEN.
RANGE LIST = YRANGE.
END YEAR.
LIST CHARACTER DIGITS = 0/9.
LIST LENGTH YLEN = 2, 4.
LIST RANGE YRANGE = IN 0/84, 1900/1984.
```

Note that a null value always passes all conditions. The only way to guarantee that the user enters data in a field is to assign the REQUIRED attribute to the field.

3.1.5.6 Edit Set Block. A set of field attributes conditionally selectable by an IF statement (see paragraph 3.23) is defined by an edit set block. Within an edit set block, all the field attribute statements are allowed except for the POSITION, DISPLAY, DEFAULT, and ARRAY statements. The edit set block starts with an EDIT SET statement and can end with an optional END EDIT SET statement.

3.1.5.7 List Definition Statements. The LIST statements define complex edit rules. The types of list statements are as follows:

- LIST CHARACTER
- LIST LENGTH
- LIST RANGE
- LIST SUBSTITUTE
- LIST TABLE

The format of the LIST statement is as follows:

LIST type list = specifications

where:

type	is a keyword denoting the kind of list being defined.
list	is the name by which this list can be referenced.
specifications	are the specifications for the list.

LIST statements can occur only within a SEGMENT block. A list's definition is local to the segment where it is defined, but any field, edit set, or condition block within that segment can reference a list by a statement of the following form:

type LIST = list.

3.2 ARRAY STATEMENT

The ARRAY statement defines a rectangular array of replications of a field. Each field of the array has the same attributes except for its row/column location on the screen. The ARRAY statement can appear only in a field block and has the following format:

ARRAY DIM*ENSION*(row,col), INC*REMENT*(offset,offset)

where:

row — specifies the number of rows in the field array. This number must be greater than zero.

col — specifies the number of columns in the field array. This number must be greater than zero.

offset — specifies the row and column increments, the number of lines or positions to advance to get to the next row or column of the array. The first offset specifies row increment. The second offset specifies column increment.

There are row*col fields in a field array. The position on the screen of the field containing an ARRAY statement is the position of the uppermost, leftmost field of the array. The name of this field is the name of the array.

You can define a one-dimensional field array by specifying only one row or column. If either is one, the corresponding increment is not used.

EXAMPLE

```
FIELD ARY1.
  POS(2,2)L3.                  Position of upper left field of array.
  ARRAY DIM(2,2), INC(1,4).    Two rows, two columns, no lines
.                              skipped between rows, one position
.                              between columns.
```

Array names are restricted in length. They must be short enough to allow an element number to be appended without exceeding the six character limit on TIFORM names. Since the highest element number equals row*col, you can easily determine how many characters it requires. For example, a 5 by 6 field array must have a name with no more than four characters to allow two characters for the 30 element array. Likewise, a 3 by 3 field array must have a name with five or fewer characters.

You use the following special notation for referencing individual fields in an array:

array(row,col)

where:

array is the name of the field array.

row is the number of the row of the field array being referenced. In a GROUP statement, you can use an * to specify all rows.

col is the number of the column of the field array being referenced. In a GROUP statement, you can use an * to specify all columns.

EXAMPLE

```
GROUP G1 = XYZ(*,*).     The group G1 includes all the fields of the field
.                        array XYZ in row order.

GROUP G2 = XYZ(2,*).     The group G2 includes all of row 2 of the field
.                        array XYZ.

GROUP G3 = XYZ(*,3).     The group G3 includes all of column 3 of the field
.                        array XYZ.

GROUP G4 = XYZ(2,3).     The group G4 includes only the one field at row 2,
.                        column 3 of the field array XYZ.
.

GROUP G5 = XYZ.          The group G5 includes only the top, leftmost field,
   or                    field XYZ.
GROUP G5 = XYZ(1,1).
.
```

3.3 AUTOSKIP AND NOAUTOSKIP STATEMENTS

The AUTOSKIP statement specifies the autoskip attribute for a field, automatically closing the field when its last character position is filled. The user does not have to use an event key such as Return, Forward Tab, Skip, or Enter to close an autoskip field, although an autoskip field can be closed by an event key.

If you do not specify the autoskip attribute, FDL assigns the noautoskip attribute to the field. This requires the user to press an event key to close the field. To remove the autoskip attribute from a field, include the NOAUTOSKIP statement in the FDL for its edit set.

These statements are valid only within a field block or an edit set block. They have the following format:

AUTO*SKIP*.
NOAUTO*SKIP*.

EXAMPLE

```
FIELD JUMP.          Start of JUMP field.
  POS(1,1)L5.        Begin field in row 1, column 1, with a length of 5.
  AUTOSKIP.          This field is closed when filled.
```

3.4 BRANCH STATEMENT

The BRANCH statement moves the cursor to a specified field. The branch occurs after all editing and processing is successfully completed on the current field. The BRANCH statement is valid only in a field block or edit set block and has the following format:

BRANCH TO iofield.

where:

iofield is the name of the field to read after the current field.

EXAMPLE

```
FIELD FIELD7.
  POS(1,7)L6.
  BRANCH TO FIELD9.        Branch to field FIELD9.
```

For conditional branching (GOTO), refer to the description of the IF statement.

3.5 CHARACTER LIST STATEMENT

The CHARACTER LIST statement is used with a companion LIST CHARACTER statement to validate the characters entered into a field. The CHARACTER LIST statement includes the name of the associated list of characters, plus an optional error message specification. A user who attempts to enter a value with characters not on the list receives an error message and must reenter the value.

The CHARACTER LIST statement is valid only in a field block, a condition block, or an edit set block. It has the following format:

CHAR*ACTER* LIST list [;DIAG*NOSTIC* = {message | literal}]

where:

list	is the name of a character list defined elsewhere in the same segment by a LIST CHARACTER statement.
message	is the name of an error message that is displayed in place of the standard error message if this test fails. (This specification is not valid in a CONDITION block.)
literal	is a character string that is displayed in place of the standard error message if this test fails.

If you omit the DIAGNOSTIC specification, a standard error message is displayed when an invalid value is entered. If you include the DIAGNOSTIC specification, the specified message or literal is displayed instead of the standard error message.

EXAMPLE

```
FIELD FIELD02.
  POS (3,4)L5.
  CHAR LIST=DIGIT;DIAG='Must be numeric.'.
.
LIST CHAR DIGIT=0..9.
```

3.6 CONDITION AND END CONDITION STATEMENTS

These statements define the beginning and end of a condition block. They have these formats:

COND*ITION* condition.
END COND*ITION* condition.

where:

condition is the name of the condition block being defined.

EXAMPLE

```
FIELD FIELD4.                 Field to demonstrate condition usage.
  POS(4,5)L6.                 Location and size.
  CHAR LIST=DIGIT.            Must be numeric.
  RANGE LIST=VALIDV.          Must be in range.
  IF COND HI ON *
     THEN GOTO FIELD7         !If the value < or = 5 go to FIELD7.
     ELSE GOTO FIELD9.
.
COND HI.
RANGE LIST=FIVEHI.            Condition is true if input is < or = 5.
END COND HI.
.
CHAR LIST DIGIT=0..9.         Validates each character.
LIST RANGE VALIDV=IN,0/9.     0-9 are valid.
LIST RANGE FIVEHI=IN,0/5.     0-5 are valid.
```

3.7 CONTROL MODE STATEMENT

The CONTROL MODE statement allows you to specify control functions for the form in its FDL. This relieves the application program from having to issue a call to the Control Functions routine when it opens the form. The ENABLE CONTROL MODE statement turns on the specified functions, while the DISABLE CONTROL MODE turns them off. Table 3-1 describes each of the control functions available.

You can use these statements only in the form block, the outermost level in the FDL block hierarchy. They have the following formats:

```
ENABLE CONTROL MODE mode... .
DISABLE CONTROL MODE mode... .
```

where:

mode — is a decimal integer between 1 and 16 that designates a specific function. You can specify a list of modes separated by commas.

The compiler issues a warning if you include CONTROL MODE statements in your FDL source.

EXAMPLE

```
FORM01,999999,02.
  ENABLE CONTROL MODE 2,6.      _Turn functions 2 and 6 on.
```

Table 3-1. Control Modes and Functions

Control Mode	Function
1	When on, inhibits the termination of a read on a Back Tab or Previous Field key when in the first field of the Read. When these keys are pressed, sounds a warning beep and positions the cursor at the first character of the first field of the Read.
2	When on, inhibits the Erase Input key; allows it to perform like Erase Field. Also clears a message from the message area of the screen immediately upon acknowledgement by the user. This mode is used by the ISGE and is not particularly useful for a general application.
3	When on, inhibits deblanking from the right for field and variable values.
4	When on, places the KSR terminal in the immediate write mode. Usually, the terminal is in the delayed write mode.
5	When on, I/O on the KSR terminal is unformatted. Usually, I/O is formatted.
6	When on and no data is entered, returns nulls instead of ASCII blanks. Blanks are the Form Executor default.
7	When on, forces a Previous Line/Next Line key to move the cursor to the first column of the selected field. If off, the Previous Line/Next Line keys leave the cursor in the same column of the screen as it started in. This mode is especially useful when dealing with columnar numeric data.
8	When on, performs symmetric processing to the application.
9	When on, supresses the Print key message to the screen.
10*	When on, allows an open extend on the 820. The user's 820 files are therefore concatenated.
11	When on, validation execution takes place when a Previous Field key is pressed to exit the first field in a read.
12	When on and Print key function is being used, inhibits the printing of header and trailer lines for screens.
13*	When on, allows an open extend on the 820. Open with rewind on 911.

Note:

* See Control Functions Program Notes in Section 5.

3.8 COPY STATEMENT

The COPY statement copies a value into a specified list of fields/variables. The FROM specification indicates the field, variable, or literal whose value is copied. The TO specification gives the destination field or variable for the value copied.

The COPY statement is only valid within a field block or edit set block. There are two formats for this statement—one for copying the value when the cursor enters the field and another for copying the value into the field after the cursor leaves:

(1) COPY FROM {field1 | variable1 | literal | *}... TO {field2 | variable2 | *}... [ON ENTRY]

(2) COPY FROM {field1 | variable1 | literal | *}... TO * ON COMPL*ETION*

where:

field1	is the name of the field that supplies the copy value.
variable1	is the name of the variable that supplies the copy value.
literal	is the literal to copy. It must be enclosed in single quotes.
field2	is the name of the field that receives the value. You cannot specify an element in an array.
variable2	is the name of a variable that receives the value.
*	stands for the current field.

EXAMPLE

```
COPY FROM 'XYZ' TO F6.          !Copy the literal XYZ to F6.

          or

COPY FROM F2 TO F3,F4.          !Copy the value of F2 to F3
                                !and the value of F2 to F4.

          or

COPY FROM 'ABC' TO *, F3.       !Copy the literal ABC to current
.                               field and F3. (Note that the
.                               exclamation is used to allow
.                               comments on the same record.)
```

3.9 DEFAULT STATEMENT

The DEFAULT statement assigns a field a default value, which is displayed in the field when the program calls the Prepare Segment routine. If you do not assign an explicit default value, a zero-length (null) value is installed in the field.

The DEFAULT statement is only valid within a field block. It has the following format:

DEFAULT = {iofield | variable | literal}

where:

iofield	is the name of an I/O field whose value is to be displayed as the default.
variable	is the name of a variable whose value is to be displayed as the default.
literal	is a value to be displayed as the default.

EXAMPLE

```
FIELD FIELD3.
  POS(4,5)L6.
  DEFAULT = FIELD1.        Default is from field FIELD1.

FIELD FIELD4.
  POS(7,8)L9.
  DEFAULT = 'Yes.'         Default is the literal, "Yes."
```

3.10 DEVICE STATEMENT

The DEVICE statement defines the type of device where the form is displayed. The DEVICE statement can appear only in the form or segment context. If the DEVICE statement appears in the form context, the segment and segment mask take the default device type. If the DEVICE statement appears in the segment context, the segment and segment mask become device dependent. You can create device-dependent versions of a segment for each type of device simply by editing the DEVICE statement. If this statement does not appear anywhere in the form, the segment executes regardless of the device in use.

The DEVICE statement has the following format:

DEVICE = type.

where:

type is one of the device types listed in Table 3-2.

Table 3-2. Device Types and Characteristics

Type	Characteristics
VDU-1	12 lines by 80 characters per line VDU
VDU-2	24 lines by 80 characters per line VDU
KSR-1	66 lines by 80 characters per line KSR
KSR-2	66 lines by 132 characters per line KSR

EXAMPLE

```
DEVICE = VDU-2.   1920 character/screen VDU.
```

EXAMPLE

```
FORM FORM01,999999,02.
  DEVICE=VDU-2.

SEGMENT MASK APPLE,CLEAR=Y.
    .
    .
SEGMENT PIE,(FORM01),APPLE.
    .
    .
SEGMENT MASK APPLE,CLEAR=Y.
    .
    .
SEGMENT PIE,(FORM01),APPLE.
  DEVICE=VDU-2.
    .
    .
```

The DEVICE statement in this example is for documentation purposes only. Both the segment mask and the segment take the default device type. They execute on any device.

The DEVICE statement in this example makes this segment (PIE) and segment mask (APPLE) device dependent.

3.11 DISPLAY STATEMENT

The DISPLAY statement specifies video attributes for a field or mask. For each video attribute, you can specify YES or NO to assign or remove that attribute to the field.

The format of the statement is as follows:

DISP*LAY* {ND | BR | GR} = {*YES* | *NO*}...

where each two-character keyword controls a particular display attribute, as follows:

Keyword	Attribute
ND	Nondisplay — Do not display the contents of the field.
BR	Bright — Display field using high-intensity.
BL	Blink — Blink cursor in field.
GR	Graphics — Display virtual graphics as graphics.

The scope of the DISPLAY statement depends on its context:

- When used within a field block, the attribute settings affect only that field's display attributes.
- When used within a segment mask or field mask block, the attribute settings remain in effect until another DISPLAY statement changes them or until the end of the block is reached.
- When used in an edit set block, the attribute settings allow the DISPLAY statement to be specified at run time.
- When used in a segment block, the attribute settings allow background text specified at the segment level to have display attributes.

If you assign the graphic attribute to a field, you allow application programs to write virtual graphic characters to it. These virtual graphic characters are translated to actual graphic characters before they are displayed. Appendix E discusses the virtual graphic characters and paragraph 1.3.16 discusses the use of the video attributes.

EXAMPLE

```
DISPLAY BR=Y.          Input characters are displayed bright.
```

EXAMPLE

```
FIELD SOCCER.
  POS(20,20)L50.
  DISP BR=Y.                      Display for field is bright.
  IF CONDITION HANDS ON PLAYER
     THEN EDIT SET=GOALIE.

EDIT SET GOALIE.
  DISP BR=N.                      High intensity is now turned off.
```

3.12 DISPLAY MASK STATEMENT

The DISPLAY MASK statement specifies the field mask to display prior to reading the field. This statement allows you to display instructions to the user who is about to enter data into a field. With POSTCLEAR, the Form Executor also clears the instructions from the screen once the field is entered correctly. The DISPLAY MASK statement is only valid within a field block or edit set block and has the following format:

DISP*LAY* MASK mask [,POSTCLEAR]

where:

mask	is the name of a field mask defined within the same segment.
POSTCLEAR	clears the mask's text from the screen once the field is entered correctly.

EXAMPLE

```
DISPLAY MASK FLDMSK,POSTCLEAR.      Mask is cleared.
```

3.13 EDIT SET AND END EDIT SET STATEMENTS

The EDIT SET and END EDIT SET statements mark the beginning and end of an edit set block. The EDIT SET statement has two equivalent formats:

(1) EDIT SET edits.
(2) EDITS edits.

The END EDIT SET statement has the following format:

END EDIT SET edits.

where:

edits is the name of the edit set being defined by this edit set block.

Assume that input for the field named FIELD6 is not required. If there is input, it is validated differently depending upon whether input for FIELD6 is a numeric value or a string. Cascaded edit sets are demonstrated.

EXAMPLE

```
FIELD FIELD6.
  POS(3,4)L5.                    Location and size.
  IF COND SOMTHN ON *
  THEN EDITS=LEVEL1.             No validation if no input.
.
COND SOMTHN.
  MIN LEN=1.                     True if there is something.
.
EDIT SET LEVEL1.                 Find out what it is.
  IF COND NUMBER ON *
  THEN EDITS = ENUMB             !(If a number, do ENUMB; if
  ELSE EDITS = ESTRNG.           a string, do ESTRNG.)
END EDIT SET LEVEL1.
.
COND NUMBER.
  CHAR LIST=DIGIT.               Must be a digit.
.
LIST CHAR DIGIT=0..9.
.
EDIT SET ENUMB.                  If a number, must be even
  RANGE LIST=LESS20.               and under 10, or anything
  TABLE LIST=EVEN.                 positive and under 20.
END EDIT SET ENUMB.
.
RANGE LIST LESS20=IN,0/19.       Value must be <20.
.
TABLE LIST EVEN=EX,1,3,5,7,9.    Odd values less than 10
.                                  are excluded.
.
EDIT SET ESTRNG.                 If string, must be one from
  TABLE LIST=STRNGS.               the specified list.
END EDIT SET ESTRUNG.
.
LIST TABLE STRNGS=IN,'AAA','BBB','CCC','DDDD'.
```

3.14 ERROR MESSAGE STATEMENT

The ERROR MESSAGE statement defines a message to replace a standard TIFORM edit error message. If an edit test fails, the user receives the new message instead of the standard message. Statements that reference an ERROR MESSAGE statement are the REQUIRED statement, the IF statement, and the various LIST statements. The ERROR MESSAGE statement is only valid within a segment block and has the following format:

ERROR MESS*AGE* message = literal

where:

message — is the name of the error message being defined.

literal — is the value of the error message, enclosed in single quotes. The size of the error message is the length of the literal.

EXAMPLE

```
ERROR MESSAGE NOTDIG = 'Only numeric characters are permitted.'.
```

An error message has all the attributes of a variable. It can be read and written by the application, and can be referenced anywhere that a variable can be referenced. The error message statement is included in the FDL syntax for clarity of documentation. The text string of the error message can be from 1 through 78 characters in length.

3.15 EXTERNAL STATEMENT

The EXTERNAL statement declares a list of names to be available to the application program for use with Read, Write, and Reset routines. Segment and group names are automatically external. Multiple EXTERNAL statements are allowed. The names referenced in an EXTERNAL statement must be defined in the segment where the EXTERNAL statement appears. The EXTERNAL statement has the following format:

EXT*ERNAL* {field | variable}...

where:

field — is the name of a field.

variable — is the name of a variable.

EXAMPLE

```
FORM FORM01,999999,02.
  .
  VARIABLE NUM='123'.
  .
  FIELD FIELD1.
    POS (1,10)L10.
  .
  EXT FIELD1,NUM.          Field, variable are externalized.
```

3.16 FIELD AND END FIELD STATEMENTS

The FIELD and END FIELD statements mark the beginning and end of a field block. The FIELD statement identifies the field and indicates that subsequent FDL statements pertain to that field. The optional END FIELD statement marks the end of a FIELD block. These statements are valid only within a field block and have the following formats:

```
FIELD field.
END FIELD field.
```

where:

field is the name of the field.

EXAMPLE

```
FIELD FIELD2.
  POS (3,4)L5.
END FIELD FIELD2.
```

3.17 FIELD MASK AND END FIELD MASK STATEMENTS

The FIELD MASK and END FIELD MASK statements mark the beginning and end of a field mask block. The FIELD MASK statement identifies the field mask, indicates that subsequent FDL statements form its definition, and specifies whether the screen is cleared before the mask is displayed. The optional END FIELD MASK statement marks the end of the field mask block. These statements are valid only within a field mask block and have the following formats:

```
FIELD MASK mask, CLEAR = {YES | NO}
END FIELD MASK mask.
```

where:

mask is the name of the field mask.

You reference a field mask with the DISPLAY MASK statement. The processing of the field mask by the Form Executor is terminal dependent. However, for all terminals except the KSR-1 and KSR-2, the field mask is displayed just before the field is read. If you are using a KSR, refer to the PROMPT statement.

EXAMPLE

```
FIELD MASK FLDMSK,CLEAR=N.    Do not clear the screen.
  M(23,1)'Press F1 for next page'.
END FIELD MASK FLDMSK.
```

3.18 FILLER STATEMENT

The FILLER statement defines a fill character for input fields in the form. The usual fill character is the underscore (_), but the FILLER statement allows you to change it to any character you prefer, including graphics. The statement has the following format:

FILLER = char [,DISP*LAY* = GR*APHICS*]

where:

char is the single character to use as a fill character within this form, segment, field, or edit set.

The scope of this statement depends on its context:

- If it appears within a field block, it defines the fill character for that field only.
- If the statement appears within an edit set block, the fill character becomes the field's default fill character. You can specify a different fill character for each field.
- If this statement appears outside of a field block, the fill character is used for all input fields in the form or segment that do not have a fill character of their own.

Note that a FILLER statement affects only I/O fields. Output fields always use the blank as a fill character. If you select an alphanumeric character such as 1, that character is used to fill any empty positions in the field. This might not be desirable.

EXAMPLE

```
FORM01,999999,02.
  FILLER = $'*$'.       The fill character is an asterisk for FORM01.
```

If you want to display a graphics character as the fill character, you must use the FILLER statement with a DISPLAY specification.

EXAMPLE

```
FORM01,999999,02.
  FILLER = 'a', DISPLAY = GR.      Filler is graphics character represented
.                                  by a.
```

3.19 FKEYS STATEMENT

The FKEYS statement associates fields with function keys. When the user presses one of the function keys listed in an FKEYS statement, the cursor moves to the field associated with that key. The FKEYS statement is valid only in a segment block and has the following format:

FKEYS = key/iofield.... .

where:

key — is one of the two-digit function key codes listed in Table 2-2 or 2-4.

iofield — is the name of the I/O field where the cursor is to go when the user presses the corresponding function key. (The prior field is closed as the cursor leaves.)

An FKEYS statement applies only to the segment where you define it. You must define all fields named in that same segment, and these fields cannot be arrays. The application program can override associations specified in an FKEYS list by using the Arm Event Keys command. See paragraph 5.6 for a discussion of the Arm Event Keys command.

EXAMPLE

```
FKEYS=01/FIELD3,04/FIELD1,40/ENDFLD.
```

3.20 FORM AND END FORM STATEMENTS

The FORM and END FORM statements mark the beginning and end of a form block. The FORM statment identifies the form block and indicates that subsequent FDL statements pertain to that form. The optional END FORM statement marks the end of a form block. These statements have the following formats:

```
FORM form [,part#][,rev#].
END FORM form.
```

where:

form — is the name of the form declared in the preceding FORM statement.

part# — is an optional, six-digit decimal part number.

rev# — is an optional, two-digit decimal revision number.

The name of a form must not conflict with the names of any of the other forms, segments, or segment masks stored in the same program file. The name of a form must begin with a letter and can consist of up to six letters, numerals, dollar signs, and dashes. The part number and revision number are optional and for documentation purposes only.

EXAMPLE

```
FORM FORM01,999990,01.
    .
    .
END FORM FORM01.    This is the last statement of the form.
```

3.21 GRAPHICS INPUT STATEMENT

The GRAPHICS INPUT statement permits the input of terminal-dependent graphics characters from the keyboard into the field. The statement is valid only within a field block or edit set block and has the following format:

GRAPHICS IN*PUT*.

EXAMPLE

```
FIELD F1_1.
  POS(5,75)L5.
  GRAPHICS INPUT.
```

3.22 GROUP STATEMENT

The GROUP statement defines a named list of fields, variables, and subgroups. These elements are read by the application program from left to right, top to bottom. The GROUP statement is valid only within a segment block and has the following format:

GROUP group = {field | variable | subgroup | array}... .

where:

group	is the name of the group being defined.
field	is the name of a field in the group.
variable	is the name of a variable in the group.
subgroup	is the name of a subgroup in the group.
array	is the name of an array element in the group. The element must be specified in the array(row,col) format. (See paragraph 3.2 for a discussion of field arrays.)

A group name is automatically available to the application for use in Read, Write, and Reset commands. You do not need to specify it as EXTERNAL. Paragraph 1.2.6 discusses the use of groups.

EXAMPLE

```
FIELD FIELD1.       Field 1.
  POS (1,10)L4.           Position and size.
.
FIELD FIELD2.       Field 2.
  POS (2,10)L4.
.
FIELD FIELD3.       Field 3.
  POS (12,20)L7.
.
GROUP GROUP1=FIELD1,FIELD2.  References two fields.
GROUP GROUP2=GROUP1,FIELD3.  References above group, third field.
```

3.23 IF STATEMENT

The IF statement allows you to specify conditional branching, termination, or attribute selection for the current field. The statement specifies a condition block to be tested. If the conditions in the block are satisfied, then the THEN portion of the IF statement is executed. If the conditions are not satisfied, the optional ELSE portion is executed. (You can specify an ELSE action without a THEN action, but such constructions can be difficult to understand. You can easily produce a clearer, logically equivalent statement using NOT and THEN.)

The IF statement can appear only within a field block or an edit set block. The common form of the statement (using THEN) has the following format:

```
IF [NOT] COND/TION condition [PREENTRY | POSTENTRY] ON {iofield1 | variable | *}
   THEN {GOTO iofield2 | TERM/NATE READ [IMMED/ATELY] | EDITS = edits}
   [ELSE {GOTO iofield2 | TERM/NATE READ [IMMED/ATELY] | EDITS = edits}]
```

where:

condition	is the name of a condition block to evaluate to determine whether to perform the THEN action or ELSE action.
iofield1	is the name of an input field against which the condition evaluated. For PREENTRY evaluation, this cannot be the current field.
variable	is the name of a variable against which the condition is evaluated.
*	represents the current field.
iofield2	is an I/O field where the cursor is to move.
edits	is the name of an edit set that provides conditional attributes for the current field.

Your choice of PREENTRY or POSTENTRY determines when the conditional branching, termination, or attribute selection takes place. PREENTRY means the condition is evaluated and any action is taken as soon as the cursor enters the field. POSTENTRY means the evaluation and action take place after the user has entered a value into the field, but before any edits have been applied. If you do not specify POSTENTRY, then PREENTRY is assumed.

If you have several IF statements in a field or edit set block, the Form Executor evaluates their conditions in the order the IF statements appear in the FDL. It takes the specified action as soon as it executes an IF-THEN (or IF-NOT-ELSE) statement with a condition that is satisfied or an IF-NOT-THEN (or IF-ELSE) with a condition that fails. Otherwise, it processes the field with its usual attributes.

If the action is GOTO, the cursor moves to the specified field. This can change the sequence in which fields are read from a group. Following the processing of the specified field, the group read continues as if the cursor had arrived in the specified field in the usual sequence for the group.

If the action is EDITS, then the specified edit set is used for the current field. Since that edit set can also include an IF statement, conditional attribute selection can cascade through several edit sets before a field's edits are finally selected. See paragraph 3.1.5.5 for a discussion of the evaluation of condition blocks.

If the action is TERMINATE READ (but not IMMEDIATELY), the current Read terminates following the specified entry processing and editing for the current field. Entry and output processing and editing for all fields included in the Read are performed. Thus, all fields must contain valid values before the data is returned to the application. The application program receives status code 02.

If the action is TERMINATE READ IMMEDIATELY, the Read terminates following the specified entry processing and editing for the current field. Thus, the current field must contain a valid value. No further editing or processing is performed and any data returned may be invalid. The application program receives status code 06.

EXAMPLE

```
SEGMENT SEG01,(FORM01),MASK01.
.
FIELD NEEDIT.
  POS(15,56)L5.
  IF COND NEMPTY ON *          !If the field is not empty, apply the
     THEN EDITS=VALDAT         !VALDAT edit set. If it is empty, apply
     ELSE EDITS=JUMP.          the edit set JUMP.
.
COND NEMPTY.                   Condition to determine if current
  MIN LEN=1.                   field is empty.
.
EDIT SET VALDAT.               Apply digit and range checks to
  CHAR LIST=DIGIT.             see that value is less than 100,
  RANGE LIST=NINNIN.           then right justify it.
  JUST R,FILL=0.
.
EDIT SET JUMP.                 Go directly to field 5.
  BRANCH TO FIELD5.
.
LIST CHAR DIGIT=0..9.          Valid characters are 0 through 9.
.
LIST RANGE NINNIN=IN,0/99.     Value must be 0 through 99.
```

EXAMPLE

```
FIELD UPE.
  POS(6,5)L9.
  IF COND ALPHA POSTENTRY ON *  !Apply condition ALPHA to this field. If
     THEN EDITS = ZNOT.         true, use edit set ZNOT. If not true,
  IF COND BETA PREENTRY ON F3   !apply condition BETA to F3. If true,
     THEN EDITS = YNOT.         use edit set YNOT. If not true, use
.                               current field's edits.
```

EXAMPLE

```
FIELD FIELD8.
  POS(3,24)L10.
  IF COND NEMPTY ON *          !Apply Condition NEMPTY to this field
     THEN GOTO FIELD3.         and if true go to FIELD3. If not true
  IF COND NFULL ON FIELD2      !apply Condition NFULL to FIELD2
     THEN GOTO FIELD5.         and if true go to FIELD5. If not true
.                              proceed to the next field.
COND NEMPTY.
  MIN LEN=1.
.
COND NFULL.
  LEN LIST = DIGITS.
```

3.24 JUSTIFY STATEMENT

The JUSTIFY statement calls for right or left justification of data in a field. You can specify the direction, fill character, decimal position, and timing for the justification. The JUSTIFY statement is valid only within a field block or edit set block and has the following formats:

JUST*IFY LEFT*, FILL*ER* = char [ON ENTRY | ON COMPL*ETION*]
JUST*IFY RIGHT*, FILL*ER* = char [,DEC*IMAL* = places] [ON ENTRY | ON COMPL*ETION*]

where:

char — specifies the character with which to fill the field's empty character positions after the justification.

places — specifies the number of digits to display to the right of the value's decimal point. You can specify decimal digits only for right justification.

RIGHT or LEFT specifies the direction of the justification. Right justification with a zero fill character is typically used for numeric data. Left justification with blank fill is most often used for character data.

ON ENTRY and ON COMPLETION are optional specifications for when the justification is to take place. ON ENTRY means that the justification occurs immediately after the user enters a value and that the result of the justification should be displayed. ON COMPLETION means that the justification occurs just before the application program receives the data and that the result of the justification is not displayed.

The Form Executor includes a special feature to support signed COBOL data types where the sign occupies the rightmost position in the value. For COBOL application programs only, a JUSTIFY RIGHT statement with an ON COMPLETION specification moves a field's sign character to the rightmost character position of the field. For other languages and specifications, the sign remains next to the high-order digit.

EXAMPLE

```
FIELD LEFT.
  POS(5,5)L5.
  JUSTIFY LEFT, FILL=' '.              Standard for string fields.
.
FIELD RIGHT.
  POS(4,4)L4.
  JUST R,FILL='0',DEC=2 ON COMPL.      A real number.
```

3.25 LENGTH LIST STATEMENT

The LENGTH LIST statement is used with a companion LIST LENGTH statement to validate the lengths of values (in characters) entered into a field. The LENGTH LIST statement includes the name of the associated list of valid lengths, plus an optional error message specification. A user who attempts to enter a value having a length not on the list receives an error message and must reenter the value.

The LENGTH LIST statement is valid only in a field block, a condition block, or an edit set block. It has the following format:

LEN*GTH* LIST list [;DIAG*NOSTIC* = {message | literal}]

where:

list	is the name of a length list defined elsewhere in the same segment by a LIST LENGTH statement.
message	is the name of an error message that is displayed in place of the standard error message if this test fails. (This specification is not valid in a CONDITION block.)
literal	is a character string that is displayed in place of the standard error message if this test fails.

EXAMPLE

```
FIELD ODDS.
  POS(15,8)L20.
  LEN LIST=PRIMES;DIAG=PRIMER.          One of several lengths.
.
LIST LENGTH PRIMES=1,2,3,5,7,11,13.       Prime lengths only.
ERROR MESSAGE PRIMER='Must enter prime length < or = 13'.
```

3.26 LIST CHARACTER STATEMENT

The LIST CHARACTER statement is used with one or more companion CHARACTER LIST statements to validate the characters entered by the user. The LIST CHARACTER statement specifies a list of acceptable characters. A user who attempts to enter a value with characters not on the list receives an error message and must reenter the value.

The LIST CHARACTER statement is valid only in a segment block and has the following format:

LIST CHAR*ACTER* list = {char | char..char | char/char}...

where:

list — is the name of the character list.

char — is a valid character for the field. The char..char and char/char formats are equivalent ways of specifying ranges of consecutive characters in ASCII order. The keyword BLANK represents the blank. Special characters other than the blank must be enclosed in single quotes.

EXAMPLE

```
LIST CHAR DECMAL=0..9,'.'. Standard numeric validation.
LIST CHAR ALPHA=A..Z.
```

3.27 LIST LENGTH STATEMENT

The LIST LENGTH statement is used with one or more companion LENGTH LIST statements to validate the lengths of values entered by the user. The LIST LENGTH statement specifies a list of acceptable lengths. A user who attempts to enter a value whose length is not on the list receives an error message and must reenter the value.

The LIST CHARACTER statement is valid only in a segment block and has the following format:

LIST LEN*GTH* list = length... .

where:

list — is the name of the length list.

length — is a positive integer that specifies the number of characters in a valid value. You can specify a list of lengths, separated by commas.

EXAMPLE

```
LIST LENGTH EVEN=2,4,6.        Valid input lengths.
```

3.28 LIST RANGE STATEMENT

The LIST RANGE statement is used with one or more companion RANGE LIST statements to validate the values entered by the user. The LIST RANGE statement specifies a list of acceptable ranges of values. Numeric ranges are validated arithmetically. Character ranges are sequenced in ASCII order. A user who attempts to enter a value not allowed by the list receives an error message and must reenter the value.

The LIST RANGE statement is valid only in a segment block and has the following format:

LIST RANGE list = {IN,|EX,}{value..value|value/value}...

where:

list — is the name of the range list.

value — is an upper or lower bound for a range. The value..value and value/value specifications are alternative ways of specifying pairs of upper and lower bounds. Within each pair, the first value listed (lower bound) must be less than the second (upper bound).

You must choose whether the list of ranges is to be inclusive or exclusive. If you specify IN (inclusive), a valid value must fall within one of the ranges on the list, including upper and lower bounds. If you specify EX (exclusive), a valid value must fall outside every range on the list, again including upper and lower bounds.

EXAMPLE

```
LIST RANGE LESS10=IN,0/9.

LIST RANGE MPY10S=EX,0/9,11/19,21/29. 10,20,30 and up are permissible.
.
```

3.29 LIST SUBSTITUTE STATEMENT

The LIST SUBSTITUTE statement is used with one or more companion SUBSTITUTE LIST statements to perform substitutions on values entered by the user. The LIST SUBSTITUTE statement specifies a list of possible values and their replacements. It is valid only within a segment block and has the following format:

LIST SUB*STITUTE* list = value/literal... .

where:

list — is the name of the substitute list.

value — is a value that the user might enter.

literal — is a replacement for the value entered.

When the user enters a value on the list, it is replaced by the substitute value. Otherwise, it is left unchanged. The timing of the replacement depends on whether the companion SUBSTITUTE LIST statement specifies replacement ON ENTRY or ON COMPLETION.

EXAMPLE

```
.
LIST SUB GENDER='M'/'MALE','m'/'MALE','F'/'FEMALE','f'/FEMALE'.
.
LIST SUB SYESNO='Y'/'YES','y'/'YES','YO'/'YES','yo'/'YES',
  'yes'/'YES','N'/'NO','n'/'NO','no'/'NO','NES'/'NO','nes'/'NO'.
```

3.30 LIST TABLE STATEMENT

The LIST TABLE statement is used with one or more companion TABLE LIST statements to validate the values entered by the user. The LIST TABLE statement specifies a list of acceptable or unacceptable values. A user who attempts to enter a value not allowed by the list receives an error message and must reenter the value.

The LIST TABLE statement is valid only in a segment block and has the following format:

LIST TABLE list = {IN, | EX,} value...

where:

list is the name of the table list.

value is a value the user might enter.

You must choose whether the list of values is to be inclusive or exclusive. If you specify IN (inclusive), a valid value appear on the list. If you specify EX (exclusive), for a value to be valid, it must not appear on the list.

EXAMPLE

```
LIST TABLE YESNO= IN,'Y','y','YES','yes','N','n','NO','no'.
```

3.31 MASK (BACKGROUND TEXT) STATEMENT

The mask (or background text) statement defines the position and content of a part of a background mask. The literal associated with the background text statement is displayed on the screen.

You can use this statement within a segment mask block, field mask block, or segment block. It has the following format:

M([row],[col]) literal

where:

row — is an integer specifying the absolute row number where this piece of background text is to reside or a signed integer defining the relative displacement of the current M statement from the position given by the previous M statement or POSITION statement. If you do not specify a row, the previous specification applies.

col — is an integer specifying the absolute column number where this piece of background text is to start or a signed integer defining the relative displacement of the current M statement to the rightmost column in the previous M statement or POSITION statement. The rightmost column specified is the last *used* column of the literal.

literal — is the background text. All of this text must be on a single screen line.

EXAMPLE

```
SEGMENT MASK EMPLOY,CLEAR=N.
  M(3,4) 'Name'.          Background text begins at row 3, column 4.
  M(,+16) 'Age'.          Background text begins at row 3, column 23.
  M(+1,4) 'Birthdate'.    Background text begins at row 4, column 4.
```

Multiple background text statements are allowed. You can use the DISPLAY statement to specify the video attributes of a piece of mask text if you do not want the defaults. For example, you can use bright display or graphics in a background mask. The DISPLAY statement must precede the background text statement. It remains in effect until the next DISPLAY statement occurs or until the end of the mask is reached.

EXAMPLE

```
SEGMENT MASK WHERE,CLEAR=N.      Begin WHERE mask without CLEAR.
  DISPLAY BR=Y.                  Background text 'TEXAS INSTRUMENTS'
  M(5,35) 'TEXAS INSTRUMENTS'.   is displayed in high intensity.
-                                Background text 'Austin, Texas' is
  M(6,35) 'Austin, Texas'.       displayed in high intensity.
```

If you use the graphics attribute, the text string must be composed of the graphics characters defined in Appendix E. These characters are non-alphanumeric and generally not available for use with system utilities such as the Text Editor.

NOTE

You can use background mask statements to display text on the bottom line of the screen. However, this line is also used for the display of error messages. When the user acknowledges the message by pressing Return, the bottom line of the screen is cleared and any background text previously displayed on this line is lost.

3.32 MINIMUM LENGTH STATEMENT

The MINIMUM LENGTH statement specifies a minimum length for values entered by the user. The statement can appear in an field block, edit set block, or condition block and has the following format:

MIN*IMUM* LEN*GTH* = length [;DIAG*NOSTIC* = {message | literal}]

where:

length — is the minimum length for value entered into this field.

message — is the name of an error message that is displayed in place of the standard error message if this test fails. (This specification is invalid in the CONDITION block.)

literal — is a character string that is displayed in place of the standard error message if this test fails.

EXAMPLE

```
FIELD SMALL.
  POS(20,1)L6.
  MIN LENGTH=1.          At least one character is required.
```

3.33 NO ENTRY STATEMENT

The NO ENTRY statement prevents the cursor from entering a field during a read. This statement differs from the OUTPUT statement in that a NO ENTRY field can have editing processes. The NO ENTRY statement can appear only in a field block or edit set block and has the following format:

NO ENTRY.

EXAMPLE

```
FIELD BLANK.
  POS(10,6)L25.
  IF CONDITION ZILCH ON F1
     THEN EDITS = E1.

EDIT SET E1.
  DISP BR = Y.
  NO ENTRY.
```

3.34 NOVALIDATE STATEMENT

The NOVALIDATE statement suppresses field validation for the current field. It is valid only in a field block or edit set block and has the following format:

NOVAL*IDATE*.

Field validation reapplies all edits just prior to sending the field's data to the application program. It declares an edit error if any field's edit fails. If ON ENTRY processing is specified for a field, it is quite possible that the processed field value no longer satisfies the field's edits. The NOVALIDATE attribute allows such a processed field value to be sent to the application anyway.

EXAMPLE

```
FIELD FORGET.
  POS(3,8)L34.
  NOVAL.
```

3.35 NUMERIC STATEMENT

The NUMERIC statement assigns a field a set of attributes commonly used for numeric data. It specifies right justification and a character list consisting of the characters 0..9. Option allows you to also include the plus sign, minus sign, decimal point, and blank.

The NUMERIC statement is only valid within a field block or edit set block and has the following format:

[SIGNED | UNSIGNED] NUMERIC [,FILL = char] [,DEC*IMAL* = places].

where:

char — is the fill character used for values shorter than the field length.

places — is the number of digits to display to the right of the decimal point.

If you include a SIGNED specification, the user can enter a sign as part of the value. You must reserve a position in the field for the sign, both on the screen and in the data returned to the application program. The value returned to the application program is always right-justified with zero-fill on the left. The placement of the sign depends on the programming language used:

- COBOL: Either + or − is returned in the rightmost position. If no sign is entered, + is returned.
- Pascal or FORTRAN: The sign is returned in the leftmost position. If no sign is entered, a zero is returned in that position.

If you specify UNSIGNED (or fail to specify either SIGNED OR UNSIGNED), the user is not allowed to enter a sign character. The field is right-justified on output and zero filled on the left.

The NUMERIC statement causes the FDL compiler to generate attributes for the field that are equivalent to those generated by the following statements:

```
JUSTIFY RIGHT, FILL=char [,DECIMAL=places] ON ENTRY.
[SCALE R, places ON COMPLETION.]
JUSTIFY RIGHT, FILL='0' ON COMPLETION.
CHARACTER LIST = list.
LIST CHARACTER list = 0..9 [,'.'] [,'+','-'], [,'char'].
```

The attributes generated by a NUMERIC statement are called soft attributes because they can be overridden by other FDL statements. For example, if you supply a SCALE statement with a different scale factor than the one generated by the FDL compiler, your scale factor would be used instead of the one generated.

- If you include a DECIMAL specification, the FDL compiler adds a DECIMAL specification to the JUSTIFY...ON ENTRY statement, includes the SCALE statement, and adds the decimal point to the character list.
- If you do not specify a fill character, the FDL compiler supplies a blank fill character in the JUSTIFY...ON ENTRY statement and includes the blank in the character list.
- If you include a SIGNED specification, the FDL compiler adds the plus and minus signs to the character list.

The character lists used are given six-character names according to the following algorithm:

First character	= U	if no SIGNED
	S	if SIGNED
Second character	= N	if not DECIMAL
	D	if DECIMAL
Third-fourth character	= xx	where xx equals the ASCII value of the fill char to > AA. For example, BLANK = 20, 20 + AA = CA.
Fifth-sixth character	= $$	

Thus the statement SIGNED NUMERIC, FILL = BLANK, DECIMAL = 2 yields the following list:

```
LIST CHAR SDCA$$=0..9,'.','+',',','-',BLANK.
```

The JUSTIFY, SCALE, and LIST statements do not appear on the FDL compiler listing. However, you can unambiguously derive the contents of the list created from the name of the list.

EXAMPLE

```
FIELD NUMBER.
  POS(-3,5)L8.                    A signed value with 2 decimal places
  SIGNED NUMERIC, DECIMAL=2.      is returned to the application.
.
FIELD NUMB02.
  POS(-2,5)L8.                    An unsigned value with the fill
  UNSIGNED NUMERIC, FILL=' '.     character is returned to the
.                                 application.
.
FIELD NUMB03.
  POS(-1,5)L8.                    A character list of 0..9 and right
  NUMERIC.                        justification for this field.
```

3.36 ORDERED GROUP STATEMENT

The ORDERED GROUP statement is similar to the GROUP statement in that it defines a group consisting of fields, variables, and subgroups which are read and written together by the interface routines. However, in an ordered group the elements of the group are read in the order given in the ORDERED GROUP statement rather than in their order on the screen. For more information on ordered groups, see paragraph 1.2.6.

The ORDERED GROUP statement can appear only in a segment block and has the following format:

ORDERED GROUP group = {field | variable | subgroup | array}...

where:

group	is the name of the group being defined.
field	is the name of a field that you want to include in the group.
variable	is the name of a variable that you want to include in the group.
subgroup	is the name of a group or ordered group that you want to include in the group being defined.
array	is the specification for an array element that you want to include in the group. The element must have the format array(row,col).

EXAMPLE

```
ORDERED GROUP GRP1=F5,F3,F7.      Read F5, F3, and F7 in order.
```

3.37 OUTPUT STATEMENT

The OUTPUT statement declares a field to be an output field. An output field can be read, written, and reset by the application. However, a read of an output field always returns blanks. If you do not include an OUTPUT statement in its field block, a field becomes input/output (I/O) field.

The OUTPUT statement is only valid within a field block and the only other statements allowed in the field block are POSITION, DISPLAY, and ARRAY. The OUTPUT statement has the following format:

OUTPUT.

EXAMPLE

```
FIELD SHOITM.              Display the selected item.
  POS (10,6)L4.
  OUTPUT.
```

3.38 PASS/FAIL STATEMENT

This statement specifies a comparison of fields or variables as an edit test. In the test, you can compare the value of an I/O field or variable to any other I/O field or variable. You use relational operators such as EQ and GT to express a relationship between the fields or variables being compared. If you specify PASS, the current field passes the edit test if the relationship is true. If you specify FAIL, the current field passed the edit test only if the the relationship is false.

The PASS/FAIL statement is valid only in a field block, edit set block, or condition block. It has the following format:

```
[PASS|FAIL] IF {iofield|variable|*} relop {iofield|variable|*}
      [;DIAGNOSTIC = {message|literal}]
```

where:

iofield — is the name of an I/O field used in the comparison.

variable — is the name of a variable used in the comparison.

* — stands for the field containing the cursor when the PASS/FAIL statement is in a condition or edit set block. (This allows you to use the same condition or edit set block for several fields.)

relop — is one of the relational operators:

EQ Equal

NE Not equal

LT Less than#

GT Greater than#

LE Less than or equal#

GE Greater than or equal#

— Allowed only for comparing numeric values.

message — is the name of an error message to be displayed in place of the standard error message if this test fails (not valid in the CONDITION block.)

literal — is a character string to be displayed in place of the standard error message if this test fails.

EXAMPLE

```
SEGMENT SEG01,(FORM01),MASK01.
.
VARIABLE STRING='INITIAL VALUE   '.
.
FIELD FIELD4.
  IF * EQ FIELD2.              Pass if this field = FIELD2.
  FAIL IF FIELD3 NE STRING;
    DIAG=BADADR.               Fail if FIELD3 is not equal to the variable,
.                              STRING.
.
ERROR MESSAGE BADADR='Addresses do not match.'
```

3.39 POSITION STATEMENT

The POSITION statement defines the position of the first character of a field. You can specify the absolute row*column position or the offset from a previously defined position. If you include an R specification, the offsets are figured relative to the position established by a prior M (field mask) statement or POSITION statement without an R specification. If you do not include an R specification, the row and column numbers are taken to be the actual row and column of the beginning of the field. See Table 3-2 for the number of rows and columns shown by various device types.

The POSITION statement is valid only in a field block and has the following format:

POS*ITION* [R] (row,col) L length.

where:

row — specifies the field's row number. If you do not include an R specification in the statement, this number represents an absolute row number. If you do include an R specification, this number is a signed integer that represents a row offset from the position established by a previous position or mask statement. A field cannot span more than a single row.

col — specifies the column number of the first (leftmost) position in the field. If you do not include an R specification in the statement, this number represents an absolute column number. If you do include an R specification, this number is a signed integer that represents a column offset from the position established by a previous position or mask statement.

length — is the length of the field in characters. A field cannot extend beyond the end of its row.

Though you can position fields on the bottom line of the screen, you should bear in mind that this line is also used to display standard error messages. When the error condition is corrected, the bottom line is cleared and fields on the bottom line are temporarily lost. If a field on the bottom line subsequently becomes the current field, its initial value or current value is displayed. If there is no initial value or current value, underscores indicating the position of the field are displayed.

EXAMPLE

```
FIELD FONE.
  POS (4,5)L6.      Six-character field at row 4, column 5.

FIELD FTWO.
  POS R(1,2)L7.     Seven-character field at row 5, column 7,
.                   relative to the start of the previous field.
```

EXAMPLE

```
SEGMENT SEGT01,(FORM01),MSK01.
  M(5,15) 'Check number:'.        Background text displayed at row 5,
.                                 column 15.
FIELD XXX.
  POS(,+1)L6.                     Six-character field displayed
.                                 at row 5, column 28.
```

3.40 PROMPT STATEMENT

On a KSR-type device, background text information can often be incomplete. To help the KSR user understand what to enter into a field, you can include a PROMPT statement in its field or edit set block. In an unformatted read, the text string provided by this statement is printed each time a read is issued for this field, prompting the user to enter the data. (See paragraph 2.9 for a complete discussion of operation with KSR-type devices.)

This statement concerns *KSR-type devices only*. If the device type is anything else, the PROMPT is ignored during form execution. The PROMPT statement is valid only in a field or edit set block and has the following format:

PROMPT = literal.

where:

literal is the text of the prompt.

EXAMPLE

```
FIELD PHONE.
  POS(8,45)L7.
  PROMPT='Telephone No.:'.
```

3.41 RANGE LIST STATEMENT

The RANGE LIST statement is used with a companion LIST RANGE statement to validate the characters entered into a field. The RANGE LIST statement includes the name of the associated list of ranges, plus an optional error message specification. The list of ranges can be inclusive or exclusive. With an inclusive list, the user must enter values that fall within the ranges listed. With an exclusive list, the user must enter values that fall outside the ranges listed. Otherwise, the user receives an error message and must reenter the value.

The RANGE LIST statement is valid only in a field block, a condition block, or an edit set block. It has the following format:

RANGE LIST list [;DIAG*NOSTIC* = {message | literal}]

where:

list	is the name of a range list defined elsewhere in the same segment by a LIST RANGE statement.
message	is the name of an error message that is displayed in place of the standard error message if this test fails. (This specification is not valid in a CONDITION block.
literal	is a character string that is displayed in place of the standard error message if this test fails.

If you omit the DIAGNOSTIC specification, a standard error message is displayed if an invalid value is entered. If you include the DIAGNOSTIC specification, the specified message or literal is displayed instead of the standard error message.

EXAMPLE

```
FIELD BETWEN.
  POS(1,2)L3.
  RANGE LIST LOWHI;DIAG='Rate code is invalid.'.
.
.
LIST RANGE LOWHI=IN,0/9,20/29.     0-9, 20-29 are valid.
```

3.42 REQUIRED AND NOTREQUIRED STATEMENTS

The REQUIRED statement specifies that the user must enter some value into the field. If the user attempts to enter a null value (zero-length value) or an all-blanks value into a required field, an edit error is declared.

If you do not include a REQUIRED statement in its field block, the user can enter null or all-blank data into the field, even if the field has attributes such as a minimum length or range list. You can use the NOTREQUIRED statement in an edit set to remove the required attribute from a field.

The REQUIRED and NOTREQUIRED statements are only valid within a field block or an edit set block. They have the following formats:

REQ*UIRED* [;DIAG*NOSTIC* = {message | literal}].
NOTREQ*UIRED*

where:

message — is the name of an error message.

literal — is a literal to be displayed as an error message.

If you omit the DIAGNOSTIC specification, the standard error message is displayed if no value is entered. If you include a DIAGNOSTIC specification, the referenced message or literal is displayed instead of the standard error message.

EXAMPLE

```
FIELD NEED.
  POS(5,2)L5.
  REQUIRED.       This is a required field; a standard message is used.

FIELD WANT.
  POS(,+8)L3.
  REQ;DIAG='You must enter a value!'.  A special message is used.
```

3.43 SAME AS STATEMENT

This statement defines the current field or edit set attributes to be the same as those of the specified field except for its position, display, and array attributes. If you include an EXCEPT FOR specification in the statement, you can then provide additional statements to assign more attributes to the current field.

The SAME AS statement is valid only in a field block or edit set block and has the following format:

SAME AS {iofield | edits | *} [EXCEPT FOR].

where:

iofield	is the name of an I/O field with the attributes you want.
edits	is the name an edit set with the attributes you want.
*	represents the current field.

You cannot use negative attributes, such as NOAUTOSKIP and NOTAB, in conjunction with SAME AS. If you use the SAME AS statement in conjunction with any other attribute statements, the SAME AS statement prevails.

You should watch for loops that can occur when the specified I/O field or edit set block contains additional SAME AS statements. Some loops are deleted by the FDL compiler, but none are deleted at run time.

EXAMPLE

```
FIELD FIELD1.
  POS (3,17)L6.
  DISPLAY BR = Y.
  REQUIRED.
FIELD FIELD2.
  POS(3,35)L6.
  SAME AS FIELD1 EXCEPT FOR.        Use the same attributes as in
  DISPLAY BR=Y,BL=Y.                FIELD1 plus BR and BL display.
```

3.44 SCALE STATEMENT

The SCALE statement allows you to apply a scale factor to numeric data, in effect multiplying or dividing the value entered by a power of ten. The SCALE statement is valid only within a field block or edit set block and has the following format:

SCALE {L | R}, places [ON ENTRY | ON COMPL*ETION*].

where:

places — is the scale factor, the number of decimal places the decimal point is moved in the operation.

Left (L) scaling moves the decimal point to the left, multiplying the value by the power of ten equal to the specified number of places. Right (R) scaling moves the decimal point to the right, dividing the value by the power of ten equal to the specified number of decimal places.

If you specify scaling ON ENTRY (or if you specify neither ON ENTRY nor ON COMPLETION), the scaling occurs immediately after data entry. If the user enters a decimal point with the value, that decimal point takes precedence and no scaling is performed. If you specify scaling ON COMPLETION, the scaling occurs just before the data is returned to the application program and the decimal point does not affect the scaling operation.

The following permits the user to enter 100.00 in the field, yet passes 10000 to the application program.

EXAMPLE

```
FIELD SHIFT.
  POS(6,+1)L8.
  SCALE R,2 ON COMPL.    Multiply by 100 on completion.
```

3.45 SEGMENT AND END SEGMENT STATEMENTS

These statements mark the beginning and end of a segment block. The SEGMENT statement marks the beginning of the block, gives it a name, and optionally lists the associated forms and segment mask. The END SEGMENT statement is optional and only marks the end of the block. They have the following formats:

```
SEGMENT segment [,(form...)][,segmask...].
END SEGMENT segment
```

where:

segment — is the name of this segment. This name must not conflict with the name of any other form, segment, or segment mask in the same program file.

form — is the optional name of an additional form to be associated with the segment. You can specify up to 10 form names in this list, separated by commas.

segmask — is the name of the segment mask to display when the application program actives the segment by calling the Prepare Segment routine.

A segment defined within a form block is automatically associated with that form and inserted into the corresponding form root in its program file. A segment defined outside of a form block must specify the name of the forms with which it is associated. If the form root already contains a segment with the name specified, that segment is replaced by the one being compiled. If that segment does not already exist in the form root, then an entry is added to the form root for the segment being compiled. If the form root does does not already exist in the specified program file, it is created.

The segment mask specification is optional. If you omit it and a segment mask is defined in the FDL file immediately prior to the segment, that mask is associated with the segment and displayed whenever the segment is prepared. If you omit the segment mask specification from the SEGMENT statement and do not define a segment mask immediately prior to the segment, no association is made and no segment mask is displayed with the segment.

EXAMPLE

```
SEGMENT SEGT01,(FORM01),MASK01.      Insert segment SEGT01 in form FORM01 and
                                     use mask MASK01.

   .
   .
   .
END SEGMENT SEGT01.                  End of the segment.
```

3.46 SEGMENT MASK AND END SEGMENT MASK STATEMENTS

These statements mark the beginning and end of a segment mask block, which describes the background mask displayed with the associated segment. The SEGMENT MASK statement gives a name to the segment mask and specifies whether to clear the screen before the segment is displayed. The optional END SEGMENT statement marks the end of the segment mask definition. These statements have the following formats:

SEGMENT MASK segmask, CLEAR = {*YES* | *NO*}.
END SEGMENT MASK segmask

where:

segmask is the name of the segment mask being defined.

The name you specify for the segment mask becomes the name of the overlay where the segment mask is stored and the name by which it is referenced on SEGMENT statements. You must compile the segment mask block with the segment that invokes it.

EXAMPLE

```
SEGMENT MASK SEGMSK,CLEAR=YES.     Clear the screen before the next display.
  M(5,10)'What is your name?'.
  M(6,10)'Do you wish to continue?'.
END SEGMENT MASK SEGMSK.           End this segment mask.
```

3.47 SUBSTITUTE LIST STATEMENT

The SUBSTITUTE LIST statement is used with a companion LIST SUBSTITUTE statement to perform substitutions on values entered into a field. When a user enters a value on the substitution list, it is replaced by corresponding substitute.

The SUBSTITUTE LIST statement includes the name of the associated list of values and their substitutes, plus an optional ON ENTRY or ON COMPLETION specification. The SUBSTITUTE LIST statement is valid only in a field block or edit set block and has following format:

SUB*STITUTE* LIST = list [ON ENTRY | ON COMPL*ETION*].

where:

list — is the name of a substitution list defined elsewhere in the same segment by a LIST SUBSTITUTE statement.

If you specify ON ENTRY substitution (or do not specify ON ENTRY or ON COMPLETION), as soon as the user enters a value on the list, it is replaced by its substitute. If you specify ON COMPLETION substitution, the replacement is made just before the data is returned to the application program and the user does not see the replacement value.

EXAMPLE

```
FIELD SHIP.
  POS(+1,-4)L9.
  SUB LIST=SYESNO.        Substitute YES, NO for Y, N.
.
LIST SUB SYESNO='Y'/'YES','N'/'NO'.
```

3.48 TAB AND NOTAB STATEMENTS

The TAB statement defines a tab stop at the first position of the current field. When the user presses the Forward Tab key, the cursor moves to the beginning of the next field that has the tabstop attribute. The NOTAB statement removes the tabstop attribute from a field. These statements are valid only in a field or edit set block and have the following formats:

TAB.
NOTAB.

EXAMPLE

```
FIELD MONEY.
  POS(6,16)L8.
  TAB.              A tab stop is defined for this field.
```

3.49 TABLE LIST STATEMENT

The TABLE LIST statement is used with a companion LIST TABLE statement to validate on data entered into a field. When a user enters a value into the field, it is checked against the values on the list. If the list is inclusive, the value must be on the list for it to pass the edit test. If the list is exclusive, the value must not be on the for it to pass the edit test.

The TABLE LIST statement includes the name of the associated list of values, plus an optional DIAGNOSTIC specification to provide an alternative to the standard error message the user receives when the data entered does not pass the test. The TABLE LIST statement is valid only in a field block, edit set block, or condition block. It has the following format:

TABLE LIST = list [;DIAG*NOSTIC* = {message | literal}].

where:

list	is the name of a table list defined elsewhere in the segment by a LIST TABLE statement.
message	is the name of an error message that is displayed in place of the standard error message if this test fails. (This specification is invalid in the CONDITION block.)
literal	is a character string that is displayed in place of the standard error message if this test fails.

EXAMPLE

```
FIELD MAYBE.
  POS(4,5)L6.
  TABLE LIST=YESNO.       Table of YES/NO entries.
  .
  .
  .
LIST TABLE YESNO=IN,'Y','YES','N','NO'.
```

3.50 TERMINATE READ STATEMENT

The TERMINATE READ statement forces the termination of the current Read. The conditions of termination depend upon whether you include the optional IMMEDIATELY specification in the statement. The TERMINATE READ statement is valid only within a field block or edit set block and has the following format:

TERM*INATE* READ [IMMED*IATELY*].

The TERMINATE READ statement terminates the current Read following any on-entry processing and editing for the current field. If you do not specify IMMEDIATELY, then any on-completion processing and editing for all fields included in the Read are performed. Thus, all fields must contain valid values before the data is returned to the application with a status code of 02. If you specify IMMEDIATELY, no on-completion processing or editing is performed. The unprocessed (possibly invalid) data is returned to the application program along with status code 06.

EXAMPLE

```
FIELD FINIS.
  POS(+1,-1)L2.
  TERM READ.
```

EXAMPLE

```
FIELD MAYBE.
  POS(8,7)L6.
  IF COND REJECT ON * THEN TERMINATE READ IMMEDIATELY.
```

3.51 VALUE STATEMENT

The VALUE statement specifies an initial value for a field. This initial value is installed and displayed whenever the field becomes the current field. If you do not assign the field an initial value, its current value is displayed unchanged whenever the cursor enters the field. The VALUE statement is valid only in a field block or edit set block and has the following format:

VALUE = {iofield | variable | literal}.

where:

iofield — is the name of an I/O field whose value is used as the initial value.

variable — is the name of a variable whose value is used as the initial value.

literal — is a literal value used as the initial value.

EXAMPLE

```
FIELD FLD9.
  POS(12,12)L11.
  VALUE = VAR.  Initial value is the value of the variable, VAR.
```

3.52 VARIABLE STATEMENT

The VARIABLE statement defines and names a variable. A variable is like a field that does not appear on the screen. It can be read, written, or reset by the application program. It serves as a place to hold data needed for edits or initial/default values. The size of the variable is the length of the literal the supplies its starting value.

The VARIABLE statement is only valid within a segment block and has the following format:

VAR*IABLE* variable = literal.

where:

variable — is the name of the variable being defined.

literal — is the value of the variable.

To access a variable from your application program, you must declare it external by using the EXTERNAL statement and include it in a read.

EXAMPLE

```
VAR STR='This is a variable.'.   This is a variable.
VAR NUM='12345.6'.               So is this.
VAR QTE='Patient''s number doesn''t match.'.

.       Two single quotes are required for one to be displayed.
```

4

Interactive Screen Generator/Editor

4.1 INTRODUCTION

The Interactive Screen Generator/Editor (ISGE) is a tool that allows you to design a segment and its mask interactively. You specify the mask by drawing the screen exactly as it should appear when the segment is executed. By responding to prompts from the ISGE you define the segment's fields, groups, lists, and variables. The ISGE then translates this information into FDL statements. You can use the ISGE to edit a segment created by the ISGE.

During the execution of the ISGE, the segment being operated on is stored in a file called the intermediate segment file. You can modify this file until you compile it. Once you compile it, you can no longer modify it.

Figure 4-1 is an overview of the flow of control in the ISGE.

The three major phases in the creation of a segment are as follows:

- Initiation phase — Begin the ISGE session by deciding whether to create a new segment or to edit an existing segment.
- Design phase — Design the segment mask and the field mask(s) and specify the field attributes.
- Termination phase — End the ISGE session by saving the segment for additional work during another session, by creating an FDL file for the completed segment, or by compiling the segment.

The simplest way to understand the capabilities of ISGE is to use it. The tutorial in paragraph 4.3 introduces you to ISGE and provides guidance for the initial use of its facilities. During this tutorial session, you create a segment by drawing a segment mask, creating field masks, specifying field attributes, and compiling the completed segment into FDL.

ISGE options and characteristics not covered by the tutorial are covered elsewhere. Paragraph 4.2 describes the design changes of ISGE, paragraph 4.4 discusses the intermediate segment file, and paragraph 4.5 explains how to make changes in a compiled segment. Appendix G provides a quick reference on the ISGE. It provides a list of all the attributes that can be assigned to the fields in a segment and a description of their functions.

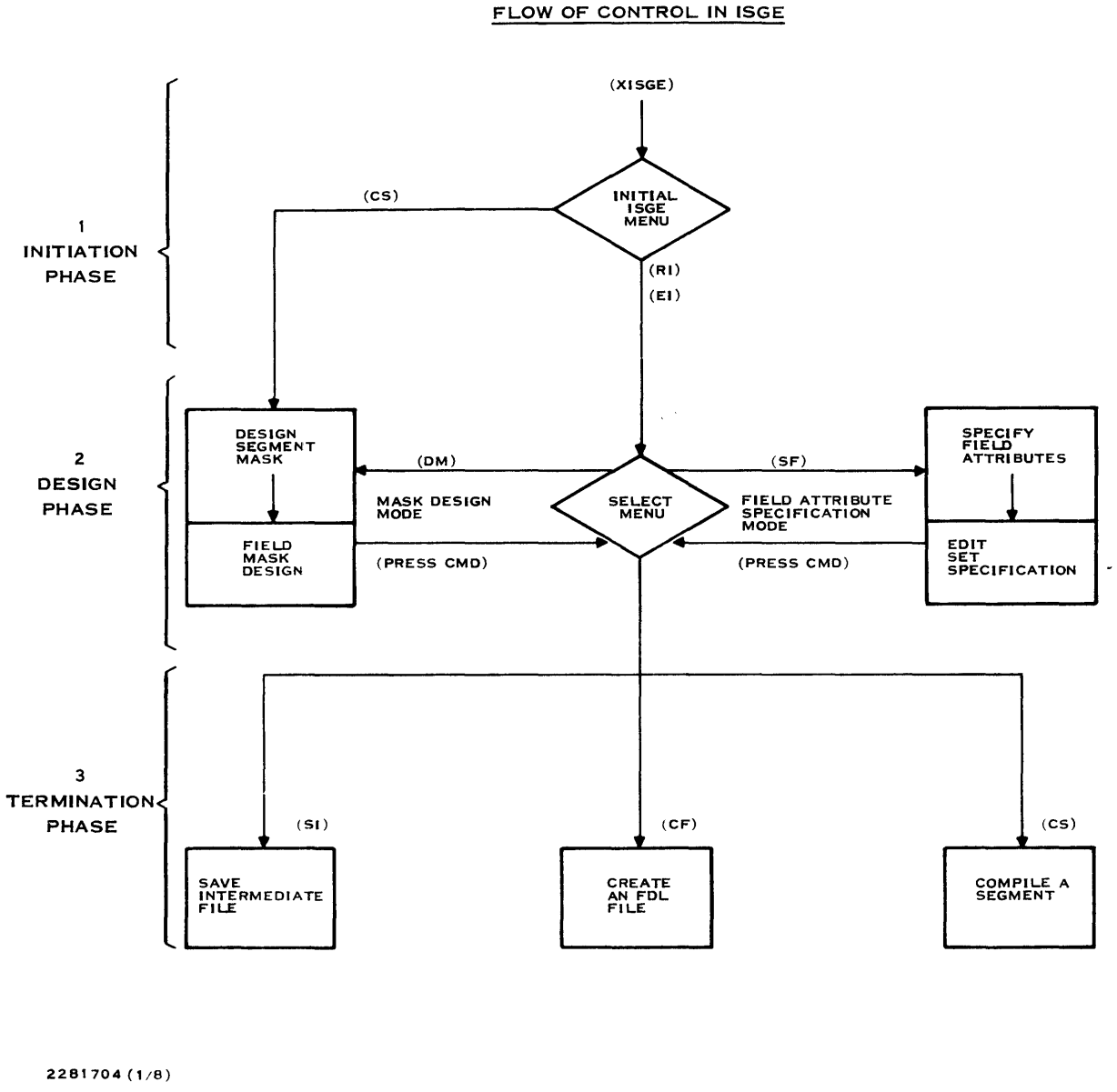


Figure 4-1. ISGE Flow of Control

4.2 ISGE DESIGN CHANGES

In previous versions of TIFORM, you could edit a segment that had already been compiled by entering the Edit a Compiled Segment (EC) command at the beginning of an ISGE session. This command took the compiled segment, decompiled it, and placed the decompiled segment in an intermediate segment file where it could then be retrieved for ISGE editing. Figure 4-2 shows an overview of this procedure. You cannot use this procedure with forms created with the current version of TIFORM. To make changes in a segment, you must save the intermediate segment file in which that segment is stored. Paragraph 4.5 discusses methods for making changes in a compiled segment.

Although the EC command remains on the initial ISGE menu, its sole use is to make 2.0 segments usable with the current release. When you use ISGE to convert 2.0 segments, the following limitations apply to the size of the segment that the ISGE can handle:

- No single variable-length item in the segment (that is, no list or group) can be longer than 576 bytes.
- The segment cannot contain more than 320 names.
- Edit sets cannot be nested more than 10 levels deep.
- The segment cannot contain more than 100 fields, excluding those for which the SAME AS attribute is specified.
- The object forms of the segment and its mask each must be less than 4000 bytes long.
- The segment cannot contain more than 100 named, user-specified error messages.

4.3 TUTORIAL: USING ISGE TO CREATE A SEGMENT

This tutorial introduces you to the Interactive Screen Generator/Editor (ISGE). During this tutorial, you learn to do the following:

- Create the directory and files needed for ISGE
- Design segments, segment masks, fields, and field masks
- Specify field attributes
- Specify edit set attributes
- Terminate an ISGE session
- Use the Form Tester utility

Figure 4-1 shows the three major phases in an ISGE session: initiation, design, and termination. This tutorial guides you through each of these phases and gives you step-by-step directions for creating the segment shown in Figure 4-3.

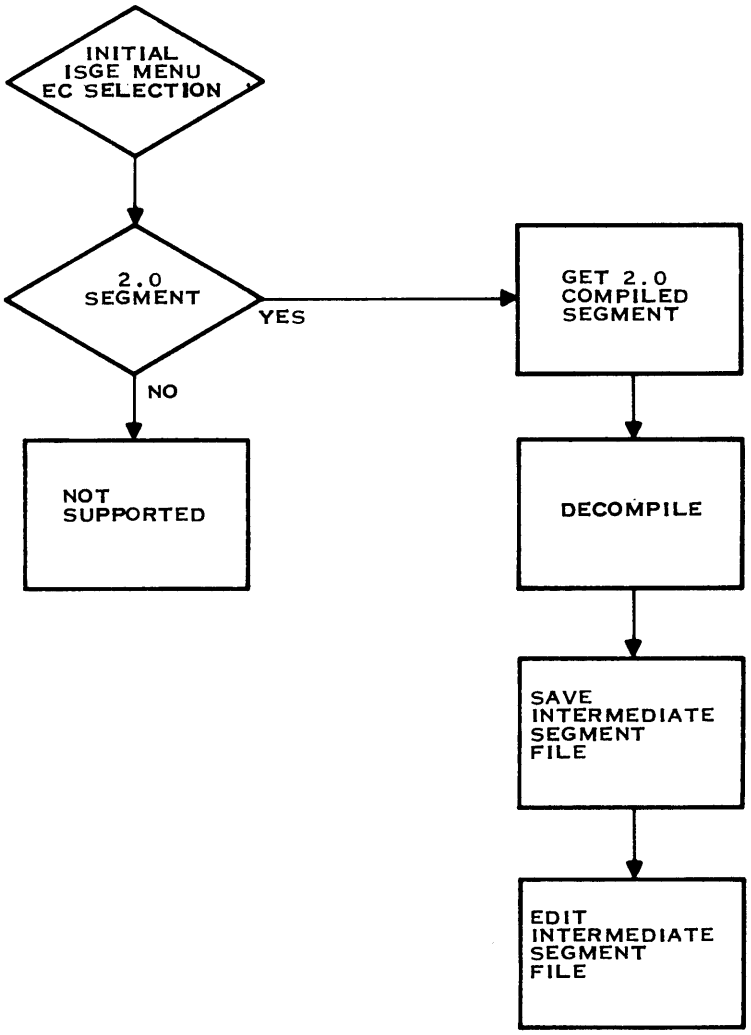


Figure 4-2. Editing a Compiled 2.0 Segment

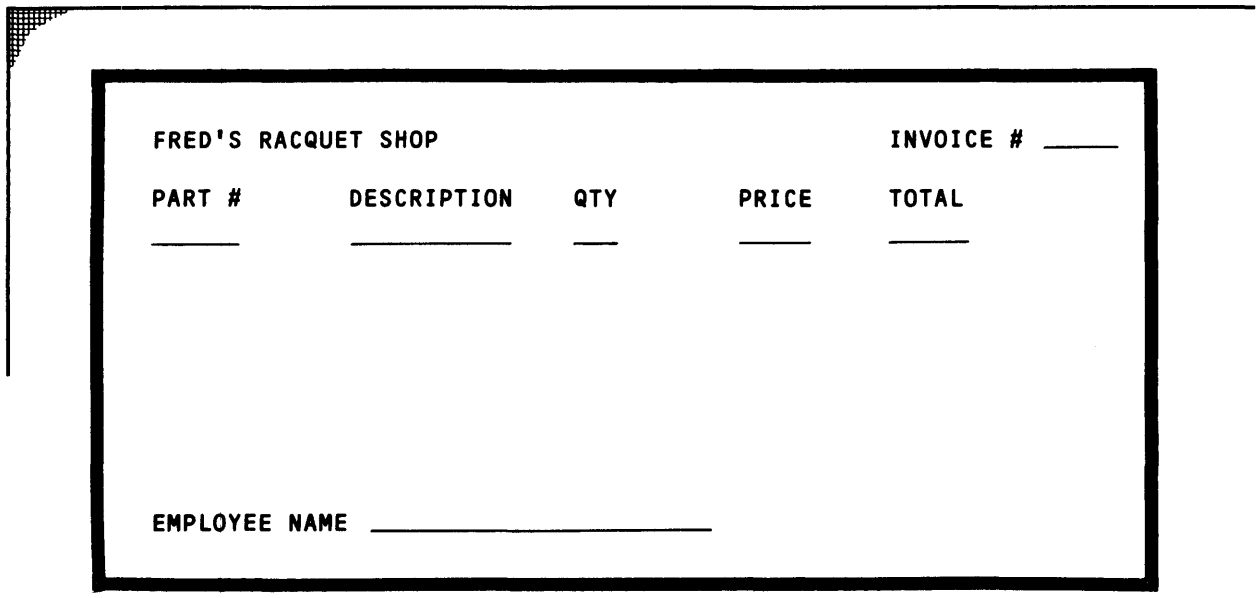


Figure 4-3. Sample Segment

This sample segment is an order entry mechanism for Fred's Racquet Shop. A clerk receives orders by telephone and uses the TIFORM segment to enter data into a COBOL program.

The completed segment performs the following functions:

- Lists the information required for each sale
- Tells the user what items can be entered for DESCRIPTION
- Displays an error message if invalid data is entered for PART #
- Copies data entered for PRICE to TOTAL
- Displays a message that tells you when a given sale earns a commission

Directions in this tutorial are written for the 931 keyboard.

NOTE

Instructions in this tutorial are indented. The keys you are to press and the prompt responses you are to enter are printed in boldface and capitalized. The key names apply to the 931 terminal. If you are using another type of terminal, refer to Appendix A. Screens depicting the results produced by your entries appear throughout the tutorial. The glossary defines the terms used in this tutorial.

4.3.1 Preparing for the Initial ISGE Session

For each segment you create, you need files for the following:

- Intermediate segment
- Source code
- Listings
- Object code

Create a directory to hold these files as follows:

Enter **CFDIR.**
Press **RETURN.**

Your screen now contains the prompts for creating a directory file. Respond to these prompts as follows:

Enter **.EXAMPLE** for PATHNAME.
Press **RETURN.**
Enter **53** for MAX ENTRIES.
Press **RETURN.**
Press **RETURN** for DEFAULT PHYSICAL RECORD SIZE.

To create a program file for ISGE, complete the following steps:

Enter **CFPRO.**
Press **RETURN.** (This displays the prompts for creating a program file.)
Enter **.EXAMPLE.PROG** for PATHNAME. (This creates a program file for the object code.)
Press **RETURN** for the remaining prompts.

To activate the ISGE and enter the initiation phase, complete the following steps:

Enter **XISGE.**
Press **RETURN.**

4.3.2 Initiation Phase

You are now in the initiation phase of ISGE (Figure 4-4).

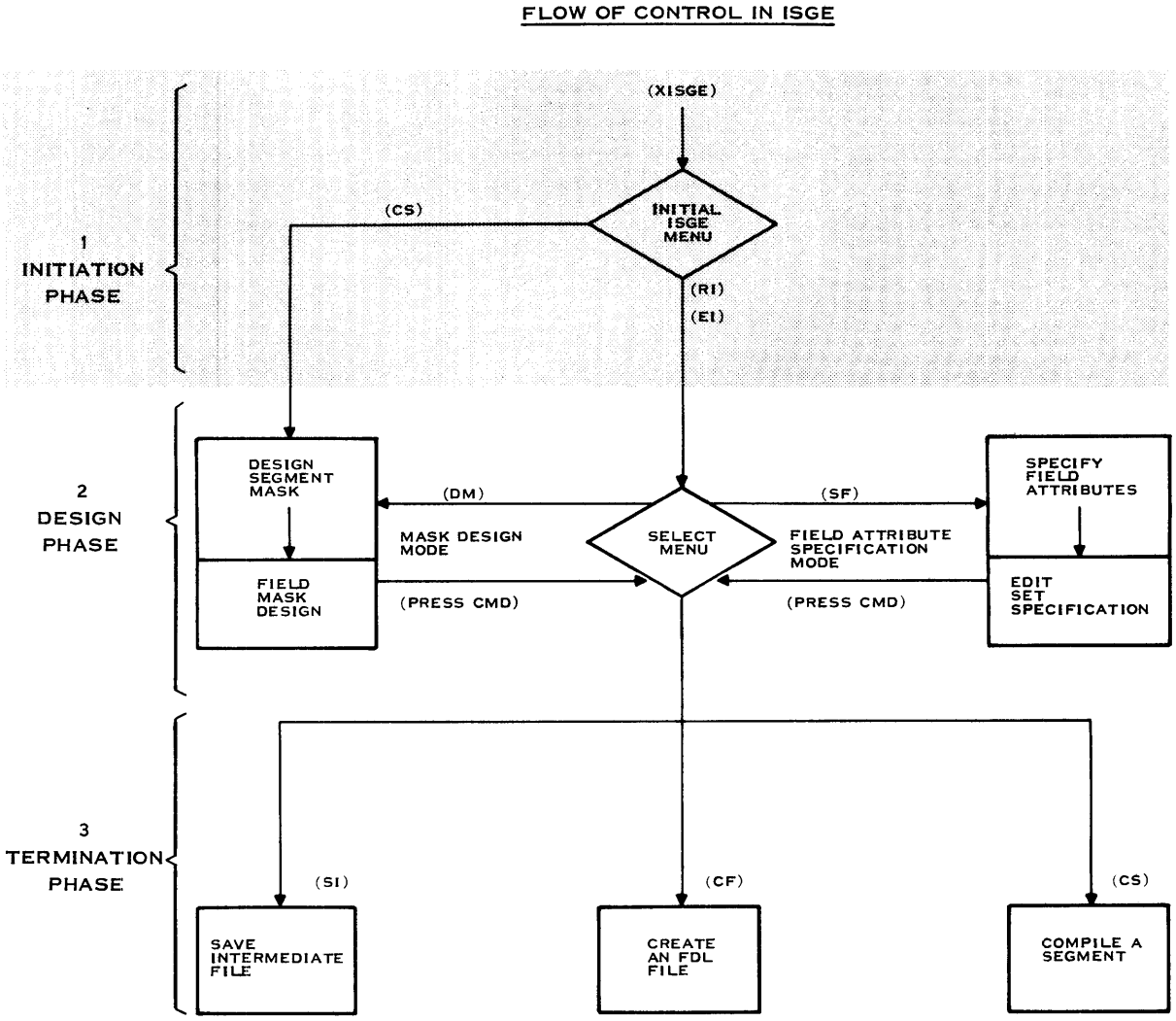


Figure 4-4. Flow of Control: Initiation Phase

The Initial ISGE Menu (Figure 4-5) appears on your screen.

During a typical ISGE session, you should select one of the five choices displayed in this menu:

- Abort this session (AB) — To terminate this session, enter AB.
- Recover an interrupted session (RI) — If your system fails while you are working on a session, you can enter RI to recover your segment upon reentry into ISGE.
- Create a new segment (CS) — To begin work on a new segment, enter CS.
- Edit an intermediate segment file (EI) — If you stored a segment in your intermediate storage file during a previous session, you can modify it by entering EI.
- Edit a segment that has been compiled (EC) — You can use this option to make 2.0 segments usable with the current release of TIFORM. If you attempt to use this command for any other purpose, an error message appears.

ISGE only recognizes uppercase characters. If you enter ab instead of AB, ISGE reports that you entered an invalid character.

In this tutorial, you will create a form composed of a single segment. (See Section 1 for a complete description of the structure of a form.) Enter the CS command in the two spaces provided at the end of the prompt display:

Enter **CS.**
Press **RETURN.**

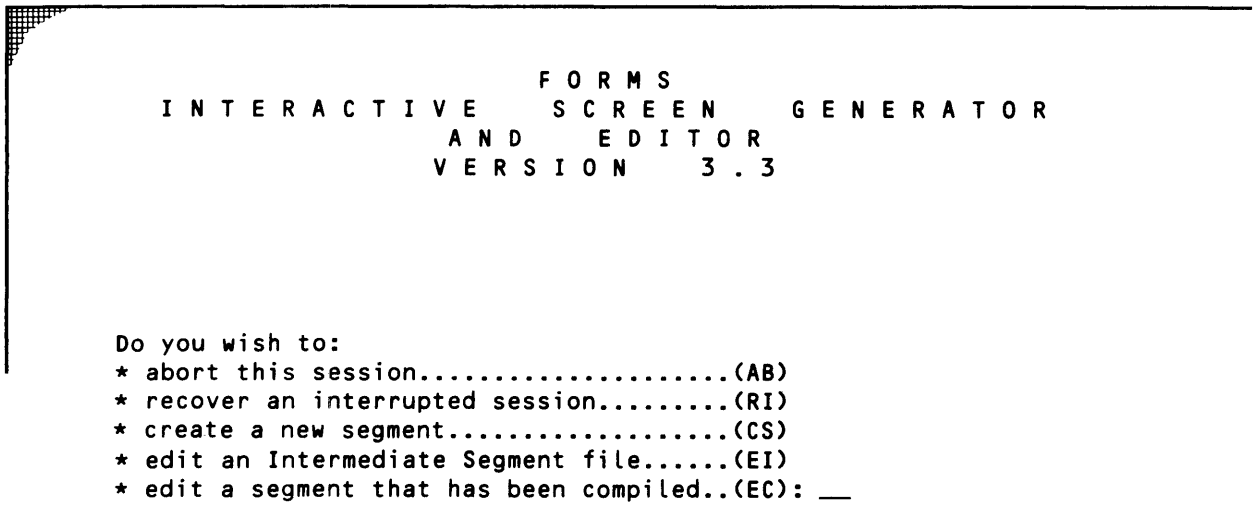


Figure 4-5. Initial ISGE Menu

The Create a New Segment ... Edit Segment Information menu (Figure 4-6) appears on your screen.

The abort option is given at the top of this menu. The cursor is initially positioned in the segment name field. To abort, use the up arrow key to position the cursor in the abort field, enter Y, and press the RETURN key.

The next three lines request names for parts of the segment you are about to create. During a typical ISGE session, you should select meaningful names for the segment, the segment mask, and the form. Each name must contain no more than six letters. Also, these names must begin with a letter. For this tutorial, enter the following names:

Enter **ORDERS** for Segment Name.
Press **RETURN.**
Enter **ORDERM** for Segment Mask Name.
Press **RETURN.**
Enter **ORDERF** for Form Name.
Press **RETURN.**

```
        Create a New Segment  ...  Edit Segment Information

                    Abort this session: N
                          Segment name: _______
                     Segment Mask Name: _______
                             Form Name: _______
                       Clear the Screen: Y
                  Segment Fill Character: _


    Select application device type (enter its number).

          DEFAULT ... execute on any possible device . (00)
          VDU-1 ..... (12 lines by  80 characters) ... (01)
          VDU-2 ..... (24 lines by  80 characters) ... (02)
          KSR-1 ..... (66 lines by  80 characters) ... (03)
          KSR-2 ..... (66 lines by 132 characters) ... (04) 00
```

Figure 4-6. Create a New Segment ... Edit Segment Information Screen (Uncompleted)

The Clear the Screen prompt asks whether the screen is to be cleared before an application uses this segment. For this tutorial accept the default (Y) as follows:

Press **RETURN.**

The Segment Fill Character prompt asks for the fill character to be used throughout this segment to represent empty field positions on the screen. The default fill character is the underscore (_). For this tutorial, accept the default as follows:

Press **SKIP.** (The RETURN key does not work here.)

Your screen should now look like Figure 4-7.

```
        Create a New Segment  ...  Edit Segment Information

                    Abort this session: N
                          Segment name: ORDERS
                     Segment Mask Name: ORDERM
                             Form Name: ORDERF
                       Clear the Screen: Y
                  Segment Fill Character: _


   Select application device type (enter its number).

         DEFAULT ... execute on any possible device . (00)
         VDU-1 ..... (12 lines by  80 characters) ... (01)
         VDU-2 ..... (24 lines by  80 characters) ... (02)
         KSR-1 ..... (66 lines by  80 characters) ... (03)
         KSR-2 ..... (66 lines by 132 characters) ... (04) 00
```

Figure 4-7. Create a New Segment ... Edit Segment Information Screen (Completed)

The lower half of the menu asks you to select an appropriate device type. Until you have completed your segment and tested it for accuracy, accept the default as follows:

Press **RETURN.**

Your screen should now look like Figure 4-8.

You can use the special function keys listed on this screen to create a segment mask or to position and define fields during the design phase of ISGE. Make a copy of the appropriate list by completing the following steps:

Press **PRINT.**
Wait for the completion message to appear.
Press **RETURN** twice.

The copy of this screen is sent to your printer.

```
              Create a New Segment  ...  Edit Segment Information

                          Abort this session: N
                                Segment name: ORDERS
                           Segment Mask Name: ORDERM
                                   Form Name: ORDERF
                             Clear the Screen: Y
                         Segment Fill Character: _

Start off by creating a segment mask and positioning fields.  The screen will
go blank as the "Mask Design" mode is entered.  The active function keys are:

          F1: Position Cursor                          F5: Copy Block
          F2: Insert Lines                             F6: Move Field
          F3: Delete Lines                             F7: Delete Field
          F4: Draw Vertical                            F8: Insert Field

                       CMD: Leave Edit Screen Mode or Abort Function
                       F10: Enter/Leave Field Mask Mode


 [Note: Press the Print key to obtain a hard copy of this screen.]
                            Press the RETURN key to continue:
```

Figure 4-8. 931 VDT Special Function Keys

NOTE

A key name followed by a slash followed by another key name indicates that you should press the first key and then simultaneously press the second key.

The entire screen should now be blank, and the cursor should be in the upper left-hand corner.

4.3.3 Design Phase

You have completed the initiation phase of ISGE and have entered the design phase (Figure 4-9). During this phase, you can design masks (mask design mode) and specify field attributes (field attribute specification mode). Appendix G provides a quick reference to the design phase of the ISGE.

If you need to quit a session before you are through, you can save your work by pressing CMD and entering SI in response to the selection menu (refer to Figure 4-13).

4.3.3.1 Mask Design Mode. Since you are creating a new segment, you automatically enter mask design mode (Figure 4-10) immediately after leaving the initiation phase.

In mask design mode, you can design two types of masks: segment masks and field masks. You can design segment masks as soon as you enter mask design mode. To design field masks, you must press F10 and use a more limited set of keys.

Segment Mask Design. You can use the graphics characters shown in Appendix E and the regular keyboard characters and special function keys to design the segment mask.

To generate graphics characters for the segment mask on the 931, 940, and Business System terminals, press and hold the ALT key while pressing the 9 key (not on the numeric keypad). To return to the standard character set, press this key sequence again. If you press CMD when you are using the graphics character set and you return to a field, the graphics character set is still in effect and you cannot enter the characters that ISGE requires until you exit the graphics mode. Also, the space bar does not work in graphics mode. If you want to erase a character, you can use the DEL CHAR key, or you can enter another character. If you want to add spaces, you must exit the graphics mode.

Pressing CTRL/S on the 931 turns the keyboard off. The keyboard buffer, within the limits of its size, holds the characters you enter when the keyboard is off. The CTRL/Q key sequence turns the keyboard back on.

During segment mask design, you use words and graphics characters to create the part of the segment that is both constant and visible during application program execution. At this time, you also specify the position and length of the fields that are visually represented on your screen by underscores (_). Figure 4-11 shows the segment mask and the field positions and lengths for the segment you create in this tutorial.

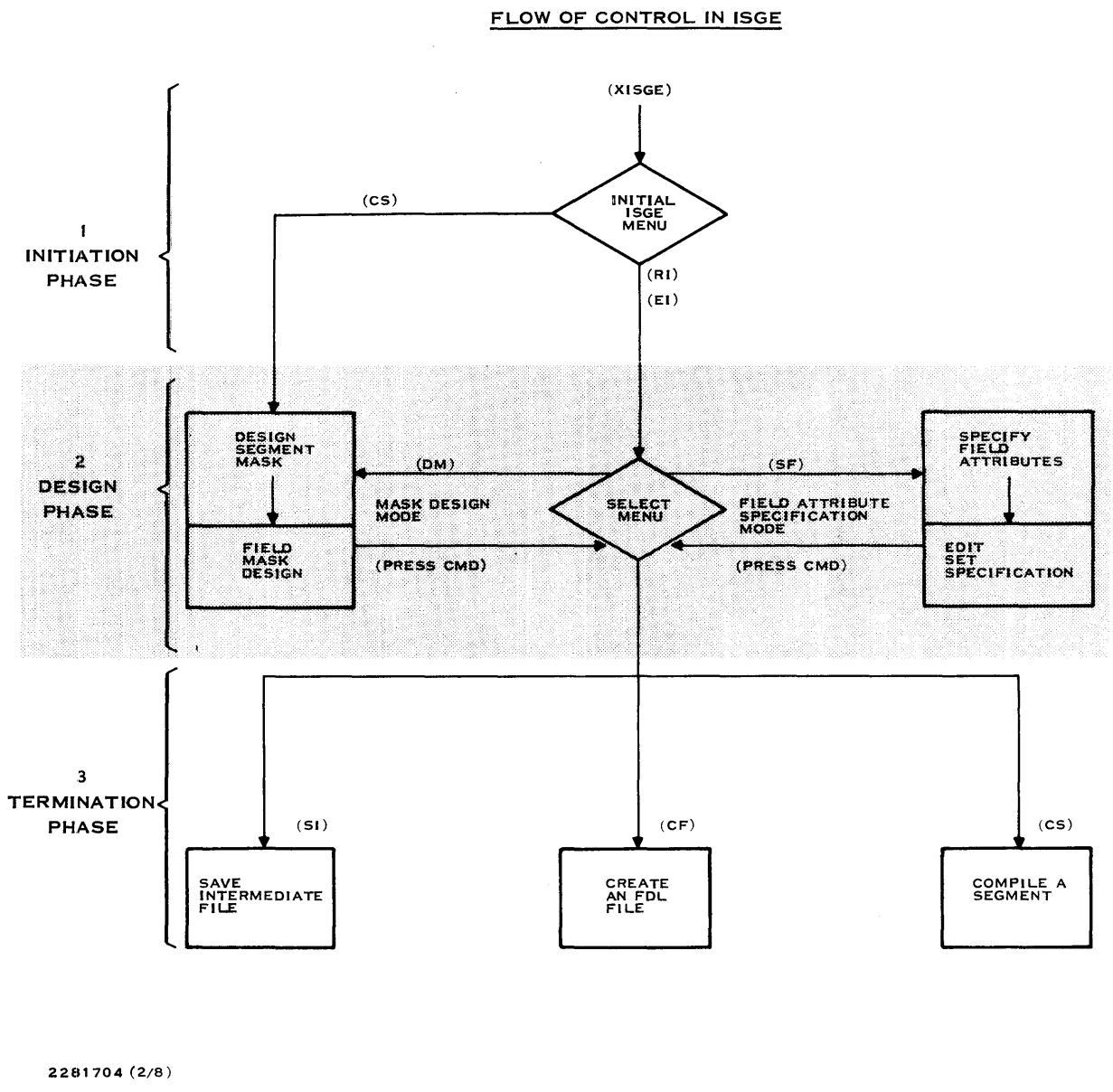


Figure 4-9. Flow of Control: Design Phase

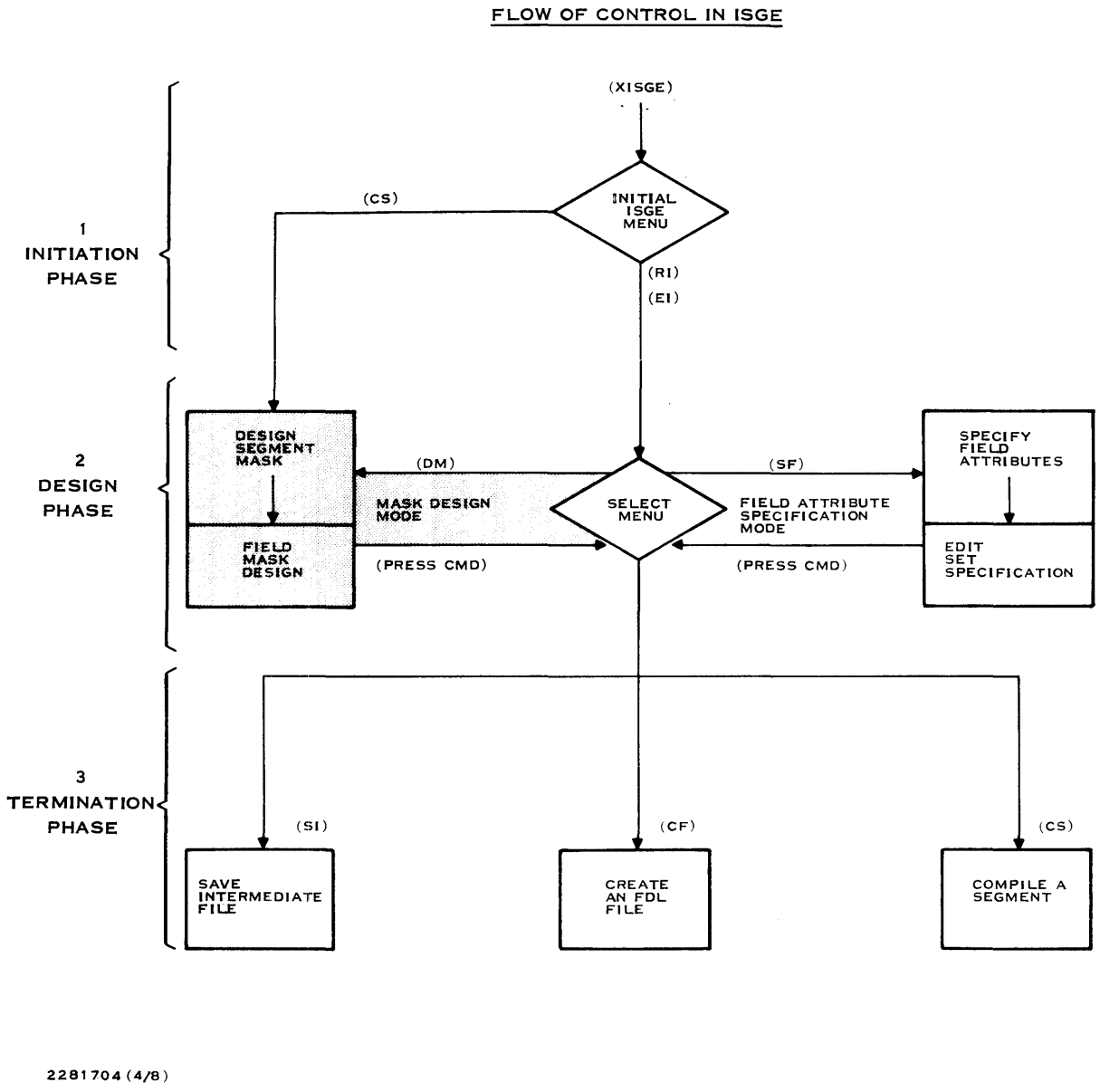


Figure 4-10. Flow of Control: Mask Design Mode

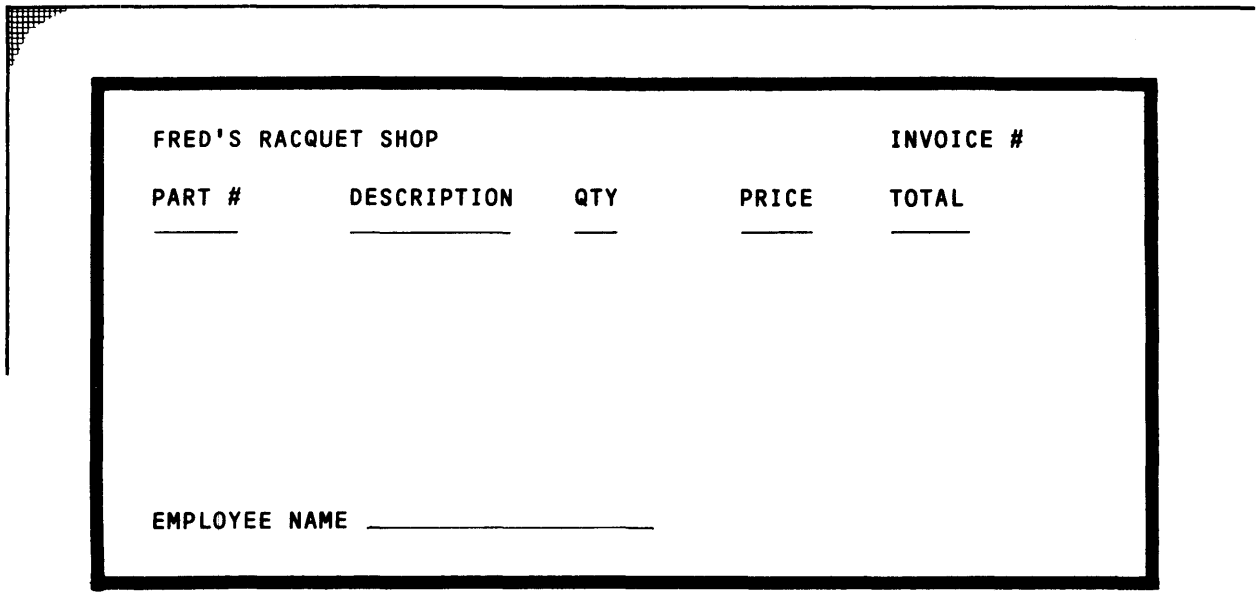


Figure 4-11. Sample Segment

The first step in designing the segment mask is to make the border that outlines the segment. To make the upper left-hand corner of the border, the cursor must be in a specific row and column:

Press **F1.**

A screen prompt giving the current position of the cursor appears on the bottom of your screen. This position should be row 01, column 01. The upper left-hand corner of the border should be in row 03, column 05. You can move the cursor to this position by entering the correct coordinates:

Enter **003** for Row.
Press **RETURN.**
Enter **005** for Column.
Press **RETURN.**

The cursor should now be in row 3, column 5. Draw the upper left-hand corner of the border by entering the following graphic character:

Press **ALT/9.** (This allows you to enter graphics mode.)
Press the **asterisk (*)** key.
Press **ALT/9.** (This allows you to exit graphics mode.)

Move the cursor to the upper right-hand corner of the screen as follows:

Press **F1.**
Enter **003** for Row.
Press **RETURN.**
Enter **075** for Column.
Press **RETURN.**

The cursor should now be in row 3, column 75. Make the upper right-hand corner of the border as follows:

Press **ALT/9.**
Press the **colon (:)** key.
Press **ALT/9.**

Move the cursor to the lower left-hand corner of the screen as follows:

Press **F1.**
Enter **020** for Row.
Press **RETURN.**
Enter **005** for Column.
Press **RETURN.**

Now, make the lower left-hand corner of the border as follows:

Press **ALT/9.**
Press the **hyphen (–)** key.
Press **ALT/9.**

Move the cursor to the lower right-hand corner of the screen as follows:

Press **F1.**
Enter **020** for Row.
Press **RETURN.**
Enter **075** for Column.
Press **RETURN.**

Make the lower right-hand corner of the border as follows:

Press **ALT/9.**
Press the **equal sign (=)** key.

You are now ready to draw the left side of the border:

Press **F4.** (This allows you to draw vertically.)

A series of prompts appears on the bottom of your screen. Answer them as follows:

Press the **right parenthesis** key after the words *Draw a vertical line using.*
Press **SKIP** to proceed to the next field. (The RETURN key does not work here.)
Press **ALT/9.**
Enter **016** for rows.

Press **RETURN.**
Enter **004** for Row. (The left side of the border begins on row 4.)
Press **RETURN.**
Enter **005** for Column. (The left side of the border fills column 5.)
Press **RETURN.**

Now, draw the right side of the border as follows:

Press **F4.**
Press **ALT/9.**
Press the **right parenthesis** key after the words *Draw a vertical line using.*
Press **ALT/9.**

Press **SKIP**. (The RETURN key does not work here.)
Enter **016** for rows.
Press **RETURN.**
Enter **004** for Row. (The right side begins on row 4.)
Press **RETURN.**
Enter **075** for Column. (The right side fills column 75.)
Press **RETURN.**

Use the up, down, left, and right arrow keys, and the HOME key to position the cursor in the space immediately to the right of the left-hand corner of the border. (You cannot use the RETURN key to position the cursor at this time since it produces a graphics character during segment mask design.) You are now ready to connect the top corners of the border with a horizontal line:

Press **ALT/9.**
Press **6** and hold until the corners are joined.

Use the up, down, left, and right arrow keys, and the HOME key to position the cursor in the space immediately to the right of the lower left-hand corner of the border. Connect the corners with a horizontal line as follows:

Press **6** and hold until the corners are joined.
Press **ALT/9.**

The border is now complete. Fill in the background text information as follows:

Press **F1.**
Enter **005** for Row.
Press **RETURN.**
Enter **009** for Column.
Press **RETURN.**
Enter **FRED'S RACQUET SHOP.**

Press **F1.**
Enter **005** for Row.
Press **RETURN.**
Enter **058** for Column.
Press **RETURN.**
Enter **INVOICE #.**

Press **F1.**
Enter **007** for Row.
Press **RETURN.**
Enter **009** for Column.
Press **RETURN.**
Enter **PART #.**

Press F1.
Enter **007** for Row.
Press **RETURN.**
Enter **022** for column.
Press **RETURN.**
Enter **DESCRIPTION.**

Press F1.
Enter **007** for Row.
Press **RETURN.**
Enter **037** for Column.
Press **RETURN.**
Enter **QTY.**

Press **F1.**
Enter **007** for Row.
Press **RETURN.**
Enter **048** for Column.
Press **RETURN.**
Enter **PRICE.**

Press **F1.**
Enter **007** for Row.
Press **RETURN.**
Enter **058** for Column.
Press **RETURN.**
Enter **TOTAL.**

Press **F1.**
Enter **018** for Row.
Press **RETURN.**
Enter **009** for Column.
Press **RETURN.**
Enter **EMPLOYEE NAME.**

Your next step is to define the length and position of each field you want to create. Insert a field four spaces long next to the word INVOICE # as follows:

Press **F8.** (This inserts a field.)
Enter **005** for the Row.
Press **RETURN.**
Enter **067** for the column.
Press **RETURN.**
Enter **004** for the Length.
Press **RETURN.**

Insert a field six spaces long beneath the PART # as follows:

Press **F8.**
Enter **008** for the Row.
Press **RETURN.**
Enter **009** for the Column.
Press **RETURN.**
Enter **006** for the Length.
Press **RETURN.**

Insert a field 11 spaces long beneath DESCRIPTION as follows:

Press **F8.**
Enter **008** for the Row.
Press **RETURN.**
Enter **022** for the Column.
Press **RETURN.**
Enter **011** for the Length.
Press **RETURN.**

Insert a field three spaces long beneath QTY as follows:

Press **F8.**
Enter **008** for the Row.
Press **RETURN.**
Enter **037** for the Column.
Press **RETURN.**
Enter **003** for the Length.
Press **RETURN.**

Insert a field six spaces long beneath PRICE as follows:

Press **F8.**
Enter **008** for the Row.
Press **RETURN.**
Enter **048** for the Column.
Press **RETURN.**
Enter **006** for the Length.
Press **RETURN.**

Insert a field seven spaces long beneath TOTAL as follows:

Press **F8.**
Enter **008** for the Row.
Press **RETURN.**
Enter **058** for the Column.
Press **RETURN.**
Enter **007** for the Length.
Press **RETURN.**

Insert a field 20 spaces long next to EMPLOYEE NAME as follows:

Press **F8.**
Enter **018** for the Row.
Press **RETURN.**
Enter **023** for the Column.
Press **RETURN.**
Enter **020** for the Length.
Press **RETURN.**

You have now completed the segment mask. Your screen should look like Figure 4-11.

Field Mask Design. You can use the graphics characters shown in Appendix E, the regular keyboard characters, and the special function keys to design field masks.

During field mask design, you create text that appears at specified positions in the application program. Typically, a field mask provides the information you need to correctly fill in a particular field or delivers a message commenting on your response. When you complete a given field, the field mask usually disappears.

Field masks associated with a specific field are displayed either upon entry into that field or when specified conditions are satisfied. The first field mask you create in this tutorial is to be associated with the field positioned below DESCRIPTION. To make this association, place the cursor on the dotted line directly beneath the D and proceed as follows:

Press **F10.**

Field mask prompts appear on the bottom of your screen:

Enter **DSCRIP** for name.
Press **RETURN.**

Field masks can be displayed in bright or normal intensity. To specify the bright attribute for field mask, enter Y in response to the BR prompt. For normal display, accept the initial value, N.

You now select the display attribute you want. For this tutorial, proceed as follows:

Enter **Y** for BR.
Press **RETURN**.

Your screen should look like Figure 4-12.

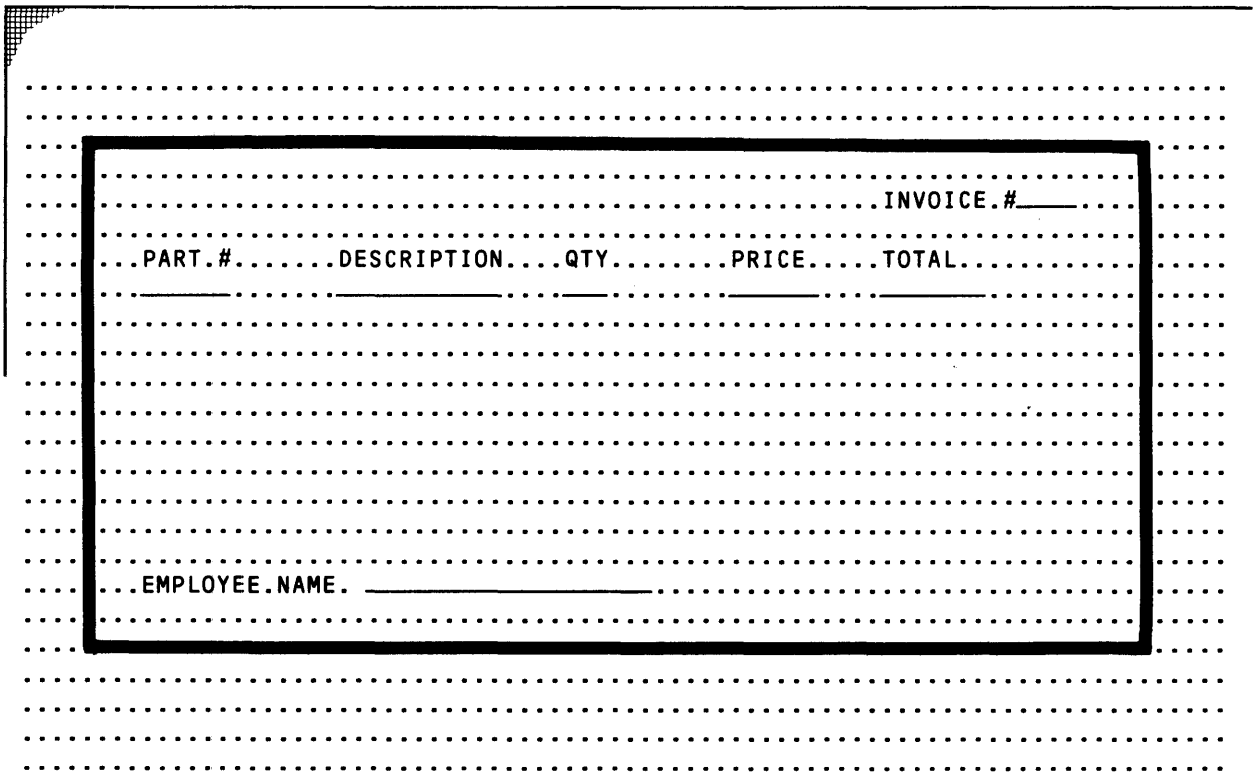


Figure 4-12. Create Field Mask

To create the field mask for the field labeled DESCRIPTION, complete the following steps:

Press **F1.**
Enter **009** for Row.
Press **RETURN.**
Enter **012** for Column.
Press **RETURN.**
Enter **DESCRIPTION ITEMS ARE:**.
Place cursor on next line beneath the D.
Enter **HAT, RACKET, CLOTHING, BALLS, SHOES.**
Press **F10.** (This displays prompts that allow you to specify what to do with the mask.)

The screen is now blank except for the Field Mask Completion prompts on the bottom line. You now decide whether to keep the mask, delete it, or abort this activity entirely. For this tutorial, proceed as follows:

Enter **1** for Option. (This option saves the field mask.)
Press **RETURN.**

The segment mask shown in Figure 4-11 should appear.

Sometimes a field mask appears only if the data entered into a field satisfies a particular set of conditions. These conditions are specified as an attribute of that field during field attribute specification and are associated with an edit set. When the conditions are satisfied, the attributes listed in the edit set appear on the screen. When an edit set controls the display of a field mask, the field mask must not be directly associated with a field. Therefore, the cursor must not be in a field when the mask is created.

You will now create two field masks that are controlled by edit sets. Specify the name of the first of these masks as follows:

Press **F1.**
Enter **010** for Row
Press **RETURN.**
Enter **045** for Column.
Press **RETURN.**
Press **F10.**
Enter **COMISS** for Name. (This names the field mask.)
Press **RETURN.**
Enter **Y** for BR.
Press **RETURN**.

Your screen should look like the screen in Figure 4-12. Now, create the field mask as follows:

Position the cursor two lines beneath the P of PRICE.
Enter **CONGRATULATIONS! YOU WILL.**
Position the cursor on the next line beneath the letter C.
Enter **RECEIVE A 10% COMMISSION.**
Press **F10.** (This displays prompts that allow you to specify what to do with the mask.)
Enter **1** for Option.
Press **RETURN.**

The segment mask shown in Figure 4-11 should appear on your screen.

You have completed the design of the field mask COMISS. Later in this tutorial (during field attribute specification), you will associate the field mask COMISS with the edit set DOTHIS.

Create the second field mask controlled by an edit set, as follows:

Press **F1.** (This positions the cursor.)
Enter **014** for Row.
Press **RETURN.**
Enter **033** for Column.
Press **RETURN.**
Press **F10.** (This displays the field mask prompts.)
Enter **TOOBAD** for Name. (This names the field mask.)
Press **RETURN.**
Enter **Y** for BR.
Press **RETURN**. (This displays the screen shown in Figure 4-11.)
Enter **BETTER LUCK NEXT TIME.**
Press **F10.** (This displays the prompts that allow you to specify what to do with the mask.)
Enter 1 for Option. (This indicates that the mask is finished.)
Press RETURN.

The segment mask shown in Figure 4-11 should appear on your screen.

You have completed the design of the field mask TOOBAD. Later in this tutorial (during field attribute specification) you will associate the field mask TOOBAD with the edit set DOTHAT.

You are now ready to leave the mask design mode and select a new activity:

Press **CMD.**

4.3.3.2 Selection Menu. The Selection menu shown in Figure 4-13 should now appear on your screen.

```
    Segment Name: ORDERS  Segment Mask Name: ORDERM    Form Name: ORDERF
Clear the screen: Y            Segment fill:         Device type: DEFAULT

          Do you wish to:
                  * abort this session.....(AB)
                  * design Segment/Field mask...(DM)
                  * specify Field Attributes....(SF)
                  * edit Segment Information....(ES)

                  * save the Intermediate File..(SI)
                  * create an FDL File.....(CF)
                  * compile the Segment.....(CS)  __
```

Figure 4-13. Selection Menu

As indicated in Figure 4-14, this menu is the central control point in an ISGE session.

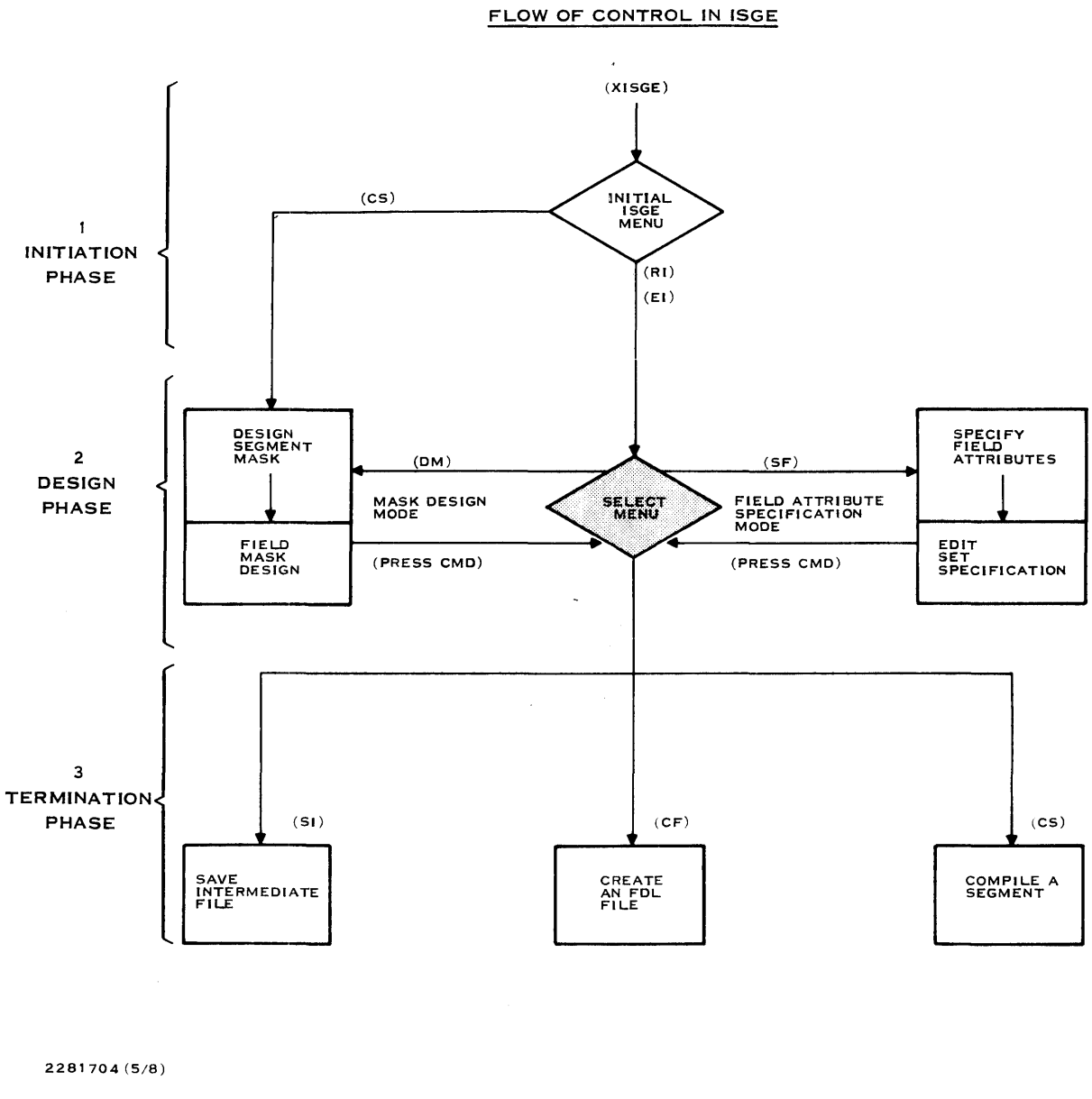


Figure 4-14. Flow of Control: Selection Menu

During the design phase of ISGE, you can move back and forth between mask design mode and field attribute specification mode by selecting design segment/field mask (DM) or specify field attributes (SF) from this menu. You can also choose to abort this session (AB), to edit segment information (ES), or to terminate this session by selecting one of the following:

- Save the Intermediate File (SI)
- Create an FDL File (CF)
- Compile the Segment (CS)

Termination options are discussed later in this tutorial in paragraph 4.3.4.

Select attributes for the fields you created during segment mask design as follows:

Enter **SF.**
Press **RETURN.**

Your screen displays the segment mask shown in Figure 4-11.

4.3.3.3 Field Attribute Specification Mode. You are now in the field attribute specification mode (Figure 4-15).

In the field attribute specification mode, you can select attributes for all of the fields in this segment. You can select those attributes from either the Field Attribute Specification (FAS) menu or the Edit Set Specification (ESS) menu. Although the segment mask appears on your screen, you cannot change it or the field masks associated with it while you are in field attribute specification mode.

You can use the regular keyboard characters and special function keys described in Table 4-1 for this activity.

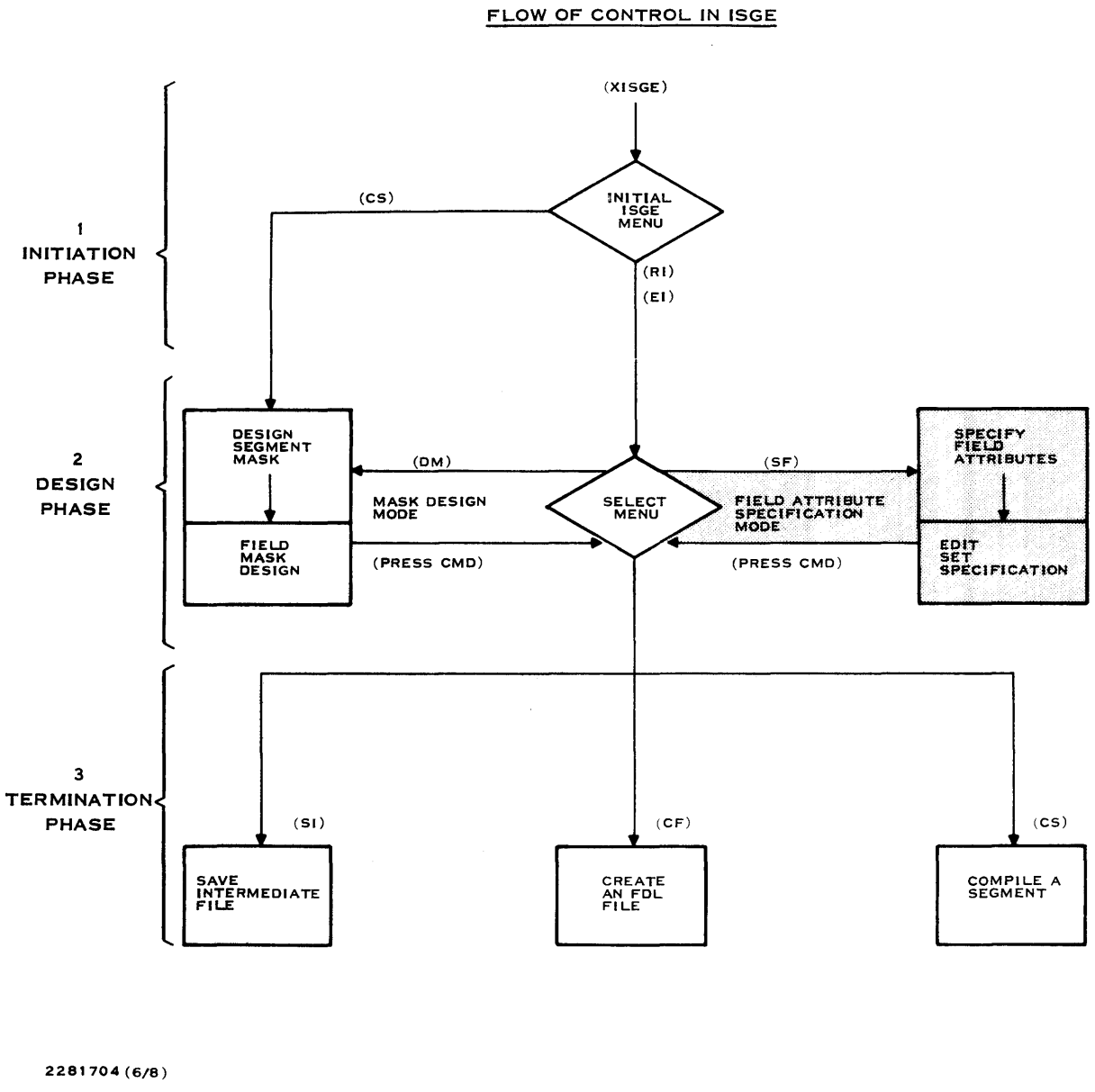


Figure 4-15. Flow of Control: FAS Mode

Table 4-1. 931 VDT Active Keys and Special Functions for FAS Mode

931 Key	Description
F1	Displays FAS menu, toggles FAS menu, and displays menu for attribute prompt area.
F2	Displays ESS menu.
F3	Moves the cursor into previous field to the left and above the current cursor position.
F4	Moves the cursor into next field to the right and below the current cursor position.
ENTER	Positions the cursor at the Complete prompt on the left side of the menu.
HOME	In top section, positions the cursor at the Name prompt. In side section, positions the cursor at the Complete prompt.
RETURN	Moves the cursor forward one field.
D, F1	When the cursor is in a field on the left side of menu and D and F1 are pressed sequentially, the association between the attribute and that field is broken.

To select attributes for a particular field, you must place the cursor in that field. Begin selecting attributes for the fields in this segment by placing the cursor in the field labeled INVOICE. Now you can select either the FAS menu or the ESS menu. For this tutorial, select the FAS menu as follows:

Press **F1.**

Field Attribute Specification Menu. The FAS menu (Figure 4-16) appears on your screen.

For a complete list of the attributes listed on both the FAS menu and the ESS menu and a description of their functions, see Appendix G. For your convenience, this tutorial defines each attribute as it is introduced.

Note that the first three attributes (row, column, and length) have been filled in with the corresponding coordinates for the field you were in when you pressed F1. They should be 005 for the row, 067 for the column, and 004 for the length.

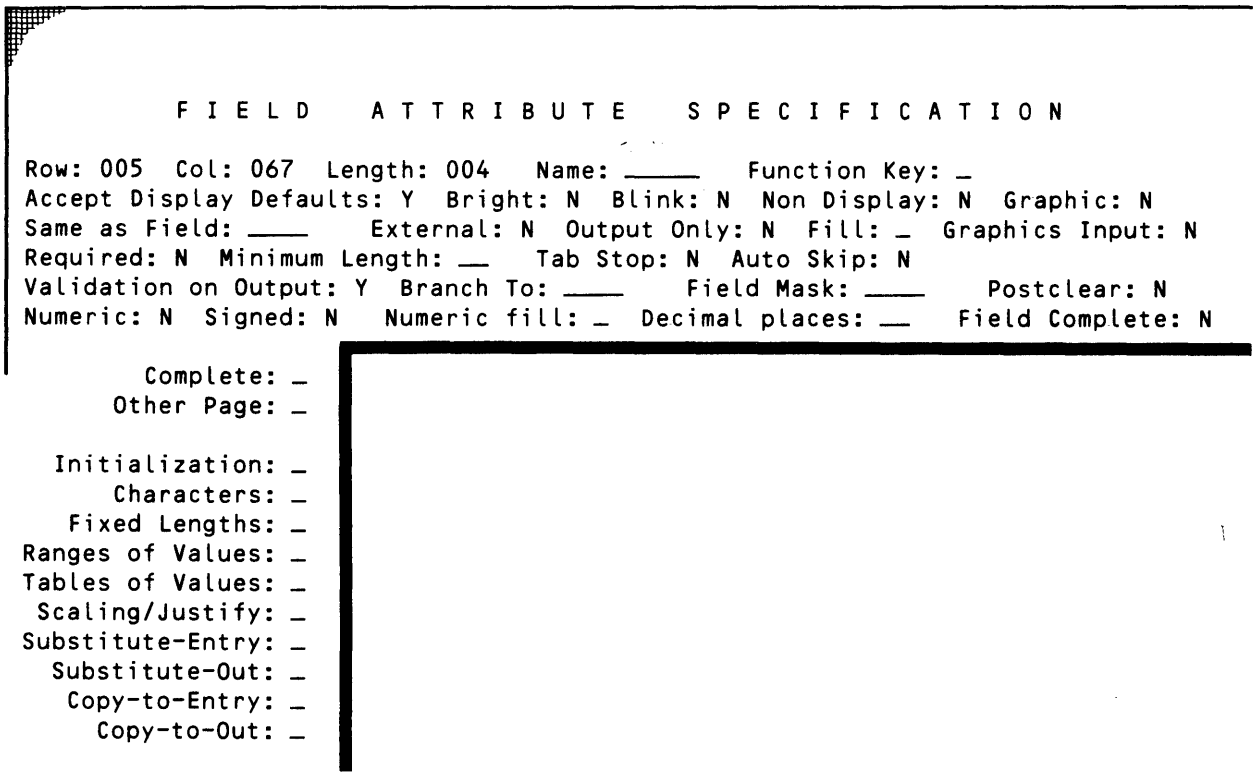


Figure 4-16. FAS Menu

Several attributes in this paragraph have default values. To accept these default values, press the RETURN key. Directions are given for the attributes you are to select. Press the right FIELD key to move through attributes for which no directions are specified. Note that you cannot press RETURN to accept the default for the FILL attribute. Instead you must press the SKIP key. For all other attributes not specified for the following edit set, tab through the attribute by pressing RETURN. Select attributes for this field as follows:

Enter **NVOICE** for Name. (This names the field.)
Enter **Y** for Required. (This specifies that you must enter data for this field.)
Enter **Y** for Numeric. (This specifies that values entered in this field must be numbers.)
Enter__for Numeric Fill.
Enter **Y** for Field Complete. (This signifies that you have completed field attribute specification for this field.)
Press **RETURN.**

Your screen looks like the screen in Figure 4-11.

You have completed the field attribute specification for INVOICE. You are now ready to select attributes for PARTN. The cursor should be in the field beneath PART #. To select attributes for this field, proceed as follows:

Press **F1.**

The FAS menu (Figure 4-16) appears on your screen. Coordinates for the row, column, and length of the field located beneath PART # appear as follows: row 008; column 009; and length 006. Complete the attribute specification for the first part of this menu as follows:

Enter **PARTN** for Name. (This names the field.)
Enter **N** for Accept Display Defaults. (This allows you to specify display attributes.)
Enter **Y** for Blink. (On a VDT, specifying this attribute causes the cursor to blink when it enters this field.)
Enter **Y** for Autoskip. (This causes the cursor to automatically leave this field after you enter data.)
Press **ENTER.** (This moves the cursor to the lower left-hand section of the FAS menu.)

The cursor should now be next to the attribute labeled Complete. Use the down arrow key to move the cursor to the attribute labeled Characters. This attribute allows you to specify what characters are valid for this field. To select this attribute, proceed as follows:

Press **F1.**

The screen in Figure 4-17 appears.

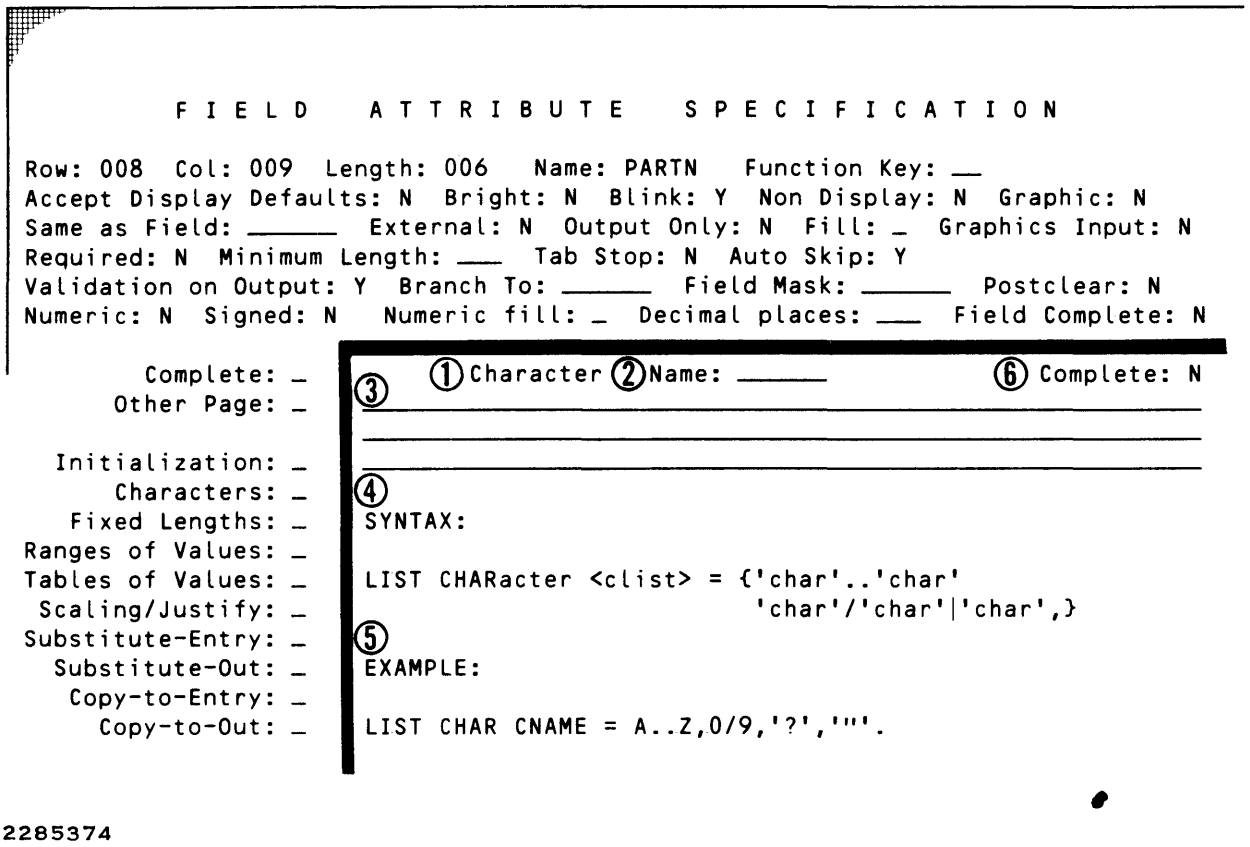
```
          F I E L D    A T T R I B U T E    S P E C I F I C A T I O N

Row: 008  Col: 009  Length: 006   Name: PARTN   Function Key: __
Accept Display Defaults: N  Bright: N  Blink: Y  Non Display: N  Graphic: N
Same as Field: ____    External: N  Output Only: N  Fill: _  Graphics Input: N
Required: N  Minimum Length: __   Tab Stop: N  Auto Skip: Y
Validation on Output: Y  Branch To: ____    Field Mask: ____    Postclear: N
Numeric: N  Signed: N   Numeric fill: _  Decimal places: __   Field Complete: N

        Complete: _     Character  Name: ____                  Complete: N
      Other Page: _     ____________________________________________________
                        ____________________________________________________
  Initialization: _     ____________________________________________________
      Characters: _
   Fixed Lengths: _     SYNTAX:
Ranges of Values: _
Tables of Values: _     LIST CHARacter <clist> = {'char'..'char'
 Scaling/Justify: _                              'char'/'char'|'char',}
Substitute-Entry: _
  Substitute-Out: _     EXAMPLE:
   Copy-to-Entry: _
     Copy-to-Out: _     LIST CHAR CNAME = A..Z,0/9,'?','"'.
```

Figure 4-17. FAS Menu — Character Prompt

The section on the right side of your screen should now contain a series of prompts. This section is called the attribute prompt area. The prompts that appear in this area differ according to which attribute you select. For this attribute, you must complete six prompts. Figure 4-18 labels these prompts, and the text that follows describes them.



2285374

Figure 4-18. FAS Menu — Character Prompt Labeled

1. The name of the attribute you have selected is printed here.
2. Insert the name of the list you are about to create.
3. Using the example shown on the screen as a guide, enter the list of characters that may be entered in this field.
4. The syntax for the FDL statements generated by this specification is shown here.
5. Look at the format of the text entered after the equal sign in this example. The character list you enter (step 3) should follow this format.
6. Enter N if you want to enter additional information; enter Y to return to the FAS menu when you have completed the prompt responses.

To respond to these prompts, complete the following steps:

Enter **DIGIT** for Name. (This names the attribute.)
Press **RETURN.**
Enter **0..9,BLANK** on the dotted line.
Press **RETURN** three times.
Enter **Y** for Complete.
Press **RETURN.**

The text in the attribute prompt area should disappear. The cursor is on the line labeled Characters. Move the cursor to the attribute labeled Fixed Lengths. This attribute allows you to specify valid lengths for data entered in this field. To select this attribute, proceed as follows:

Press **F1.**

Respond to the prompts in the attribute prompt area as follows:

Enter **PNLEN** for Name. (This names the attribute.)
Press **RETURN.**
Enter **4** on the dotted line.
Press **RETURN** three times.
Enter **Y** for Complete.
Press **RETURN.**

The cursor is now on the line labeled Fixed Lengths. Move the cursor to the line labeled Other Page. This attribute allows you to gain access to another page of attributes. Respond as follows:

Press **F1.**

Note that new attributes appear in the list below Other Page. The cursor is next to Complete. Use the down arrow key to move the cursor next to User Error Msg. Select this attribute to create your own error messages:

Press **F1.**

A series of prompts appear in the attribute prompt area. During a typical ISGE session, you select the type of user-defined error message you want. For this tutorial, use the down arrow key to move the cursor next to Length List. Then proceed as follows:

Press **F1.**

Respond to the prompts that appear as follows:

Enter **ERRPN** for Name. (This is the name of the list.)
Press **RETURN.**
Enter **PART NUMBER MUST BE 4 DIGITS** on the dotted line.
Press **RETURN** two times.
Enter **Y** for Complete.
Press **RETURN.** (The attribute prompt display disappears and the field attribute specification display appears.)
Move the cursor to Complete.
Press **F1.** (This returns the cursor to the Field Complete prompt.)
Enter **Y** for Field Complete.
Press **RETURN.**

Your screen now looks like the one in Figure 4-11. The cursor is in the field located beneath DESCRIPTION. To select attributes for this field, proceed as follows:

Press **F1.**

The FAS menu appears on your screen. Coordinates for the row, column, and length of the field appear as follows: row, 008; column, 022; and length, 011. Note that the Field Mask attribute contains the name DSCRIP, which is the field mask associated with this field during mask design mode. To complete the field attribute specification for this field, proceed as follows:

Enter **DESCRP** for Name. (This names the field.)
Accept the default, **Y,** for Postclear. (This clears the mask from the screen after the cursor leaves the field.)
Press **ENTER** (This moves the cursor to the second section of the FAS menu.)
Move the cursor to Tables of Values.

The Tables of Values attribute allows you to specify a list of values that are valid or invalid for this field. The specified table can be inclusive (IN), which means that any value entered in this field must match a value in the table, or exclusive (EX), which means that any value entered must not be in the listed table. Complete the prompts for this attribute as follows:

Press **F1.**
Enter **THINGS** for Name. (This names the attribute.)
Press **RETURN.**
Enter **IN, 'HAT',' RACKET', 'CLOTHING', 'BALLS', 'SHOES'** on the dotted line.
Press **RETURN** three times.
Enter **Y** for Complete.
Press **RETURN.**
Move the cursor to Complete.
Press **F1.** (This moves the cursor to Field Complete.)
Enter **Y** for Field Complete.
Press **RETURN.**

The segment mask shown in Figure 4-11 appears on your screen. The cursor is in the field located beneath QTY. To display the FAS menu for this field, proceed as follows:

Press **F1.**

The row, column, and length values for this field should appear as follows: row, 008; column, 037; and length, 003. Complete the following steps:

Enter **QNTITY** for Name. (This names the field.)
Enter **N** for Accept Display Defaults.
Enter **Y** for Bright. (This highlights the value.)
Press **ENTER** (This moves the cursor to Complete.)
Move the cursor to Ranges of Values.

The Ranges of Values attribute allows you to specify a list of ranges that are valid or invalid for this field. The ranges specified can be inclusive (IN), which means that any data entered into this field must be within the ranges listed, or exclusive (EX), which means that any data entered into this field must not be within the ranges listed. Complete the field attribute specification for this field as follows:

Press **F1.**
Enter **RGEQTY** for Name. (This names the attribute.)
Press **RETURN.**
Enter **IN,1/99999** on the dotted line.
Press **RETURN** three times.
Enter **Y** for Complete.
Press **RETURN.**
Move the cursor to Complete.
Press **F1.**
Enter **Y** for Field Complete.
Press **RETURN.**

The segment mask shown in Figure 4-11 appears on your screen. The cursor should be in the field located beneath PRICE.

Press **F1.**

The FAS menu appears on your screen. The values for row, column, and length appear as follows: row, 008; column, 048; and length, 006. Complete the following steps:

Enter **PRICE** for Name. (This names the field.)
Enter **N** for Accept Display Defaults.
Enter **Y** for Bright.
Enter **Y** for Numeric. (This specifies that you can enter only numbers in this field.)
Enter **Y** for Signed. (This allows you to enter signed numbers in this field and to pass them to the application.)
Enter **0** for Numeric Fill. (This fills the field with zeros.)
Enter **002** for Decimal Places. (This specifies that the values entered in this field must have two decimal places.)
Press **ENTER.**
Move the cursor to Copy-to-Entry.

The Copy-to-Entry attribute copies the field's value into another field or variable. Make the following entries:

Press **F1.**
Enter **TOTAL** for copy to field/variable. (This names the field that will receive the values entered in PRICE.)
Press **RETURN.**
Enter **Y** for Complete.
Press **RETURN.**
Move the cursor to Characters.

Press **F1.**
Enter **MONEY** for Name. (This names the character list.)
Press **RETURN.**
Enter **0..9,'.',' + ',BLANK** on the dotted lines.
Press **RETURN** three times.
Enter **Y** for Complete.
Press **RETURN.**
Move the cursor to Complete.

Press **F1.**
Enter **Y** for Field Complete.
Press **RETURN.**

The segment mask shown in Figure 4-11 reappears on your screen. The cursor is now in the field located beneath TOTAL. Proceed as follows:

Press **F1.**

The FAS menu appears on your screen. The values for row, column, and length appear as follows: row, 008; column, 058; and length, 007. Begin the field attribute specification for this field as follows:

Enter **TOTAL** for Name.
Enter **N** for Accept Display Default.
Enter **Y** for Bright.
Enter **Y** for External. (This allows the application program to refer to the field in Read, Write, and Reset commands. Section 5 describes these commands.)
Press **ENTER.**
Move the cursor to Other Page.
Press **F1.**
Move the cursor to Cond. Attributes. (This allows you to specify the conditions that determine which edit set is used with this field.)
Press **F1.**

The attribute prompts for this attribute appear on your screen. Proceed as follows:

Enter **BONUS$** for Condition. (This names the condition.)
Press **RETURN.**
Enter **PRICE** for Field/Variable.
Press **RETURN.**
Enter **DOTHIS** for then edit set is. (The edit set DOTHIS will be used if the condition is true.)
Press **RETURN.**
Enter **DOTHAT** for else edit set is. (The edit set DOTHAT will be used if the condition is false.)
Press **RETURN** three times to go to Ranges of Values Name.
Enter **RANGE$** for Ranges of Values Name. (This gives the range of values for the condition BONUS$.)
Press **RETURN** five times to go to Complete.
Enter **Y** for Complete.
Press **RETURN.**
Move cursor to Other Page.
Press **F1.**
Move cursor to Ranges of Values.
Press **F1.**
Enter **RANGE$** for Name. (This names the range list for RANGE$.)
Press **RETURN.**
Enter **IN,50/99999** on dotted line. (This specifies that if data entered in the field TOTAL is between 50 and 99999 then DOTHIS applies; otherwise, DOTHAT applies.)
Press **RETURN** three times.
Enter **Y** for Complete.
Press **RETURN.**
Move cursor to Complete.

Press **F1.**
Enter **Y** for Field Complete.
Press **RETURN.**

The segment mask shown in Figure 4-11 appears on your screen. The cursor should be in the field located next to EMPLOYEE NAME. Begin attribute specification for this field as follows:

Press **F1.**

The row, column, and length values for this field appear as follows: row, 018; column, 023; and length, 020. Complete the attribute specification for this field as follows:

Enter **YRNAME** for Name. (This names the field.)
Enter **Y** for Required.
Press **ENTER.**
Move the cursor to Characters.

Press **F1.**
Enter **YOUWHO** for Name. (This names the character list.)
Press **RETURN.**
Enter **A..Z, BLANK** on the dotted line.
Press **RETURN** three times to go to Complete.
Enter **Y** for Complete.
Press **RETURN.**
Go to Complete.
Press **F1.**
Enter **Y** for Field Complete.
Press **RETURN.**

Edit Set Specification Menu. You are now ready to create an edit set for this field. The cursor can be anywhere on the screen for this activity. An edit set contains one or more attributes that are applied to a field when the data entered in that field meets conditions specified in a conditional attribute statement. You use the Edit Set Specification (ESS) menu to specify attributes for an edit set. To display this menu, proceed as follows:

Press **F2.**

Note that the cursor need not be in a field when you specify an edit set for that field.

The ESS menu (Figure 4-19) appears on your screen.

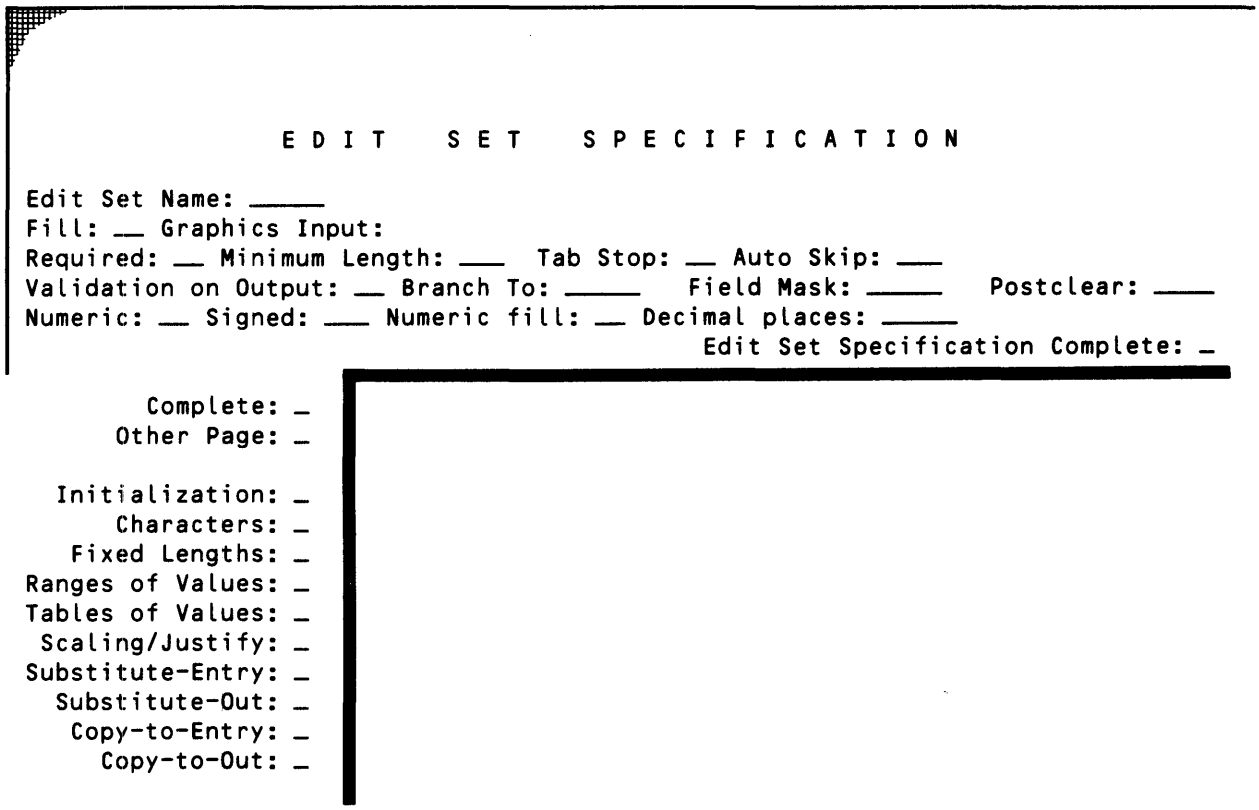


Figure 4-19. ESS Menu

The ESS menu contains many of the same attributes listed in the FAS menu. It does not contain the Position or Display attributes. Although the Array attribute is listed on the ESS menu, you cannot specify it for an edit set.

Note that you cannot press RETURN to accept the default for the FILL attribute. Instead you must press SKIP. For all other attributes not specified for the following edit set, tab through the attribute by pressing RETURN.

Create the edit set DOTHIS and associate it with the field mask COMISS as follows:

Enter **DOTHIS** for Edit Set Name
Enter **COMISS** for Field Mask.
Enter **Y** for Postclear.
Enter **Y** for Edit Set Specification Complete.
Press **RETURN**.

The segment mask shown in Figure 4-11 appears on your screen. Create the edit set DOTHAT and associate it with the field mask TOOBAD as follows:

Press **F2.**
Enter **DOTHAT** for Edit Set Name.
Enter **TOOBAD** for Field Mask.
Enter **Y** for Postclear.
Enter **Y** for Edit Set Specification Complete.
Press **RETURN**.

The segment mask shown in Figure 4-11 appears on your screen. You have now completed the design phase of this ISGE session. Enter the termination phase as follows:

Press **CMD.**

The Selection menu shown in Figure 4-13 appears on your screen.

4.3.4 Termination Phase

Figure 4-20 highlights the termination phase of ISGE.

Note that you can terminate an ISGE session in any of three ways:

- Enter SI to save the intermediate segment file created during this session.
- Enter CF to translate the intermediate segment file to FDL statements.
- Enter CS to translate the intermediate segment file and compile the resulting FDL program.

For this tutorial, use CS as follows:

Enter **CS.**
Press **RETURN.**

Prompts requesting the names of four files appear on your screen. Enter the names of the four files you created at the beginning of this tutorial:

Enter **.EXAMPLE.IMS** for Intermediate Segment File Pathname.
Press **RETURN.**
Enter **.EXAMPLE.FDLSRC** for FDL Source File Pathname.
Press **RETURN.**
Enter **.EXAMPLE.PROG** for Form Program File Pathname.
Press **RETURN.**
Enter **.EXAMPLE.LIST** for Listing File Pathname.
Press **RETURN.**
Enter **Y** for Complete.
Press **RETURN.**

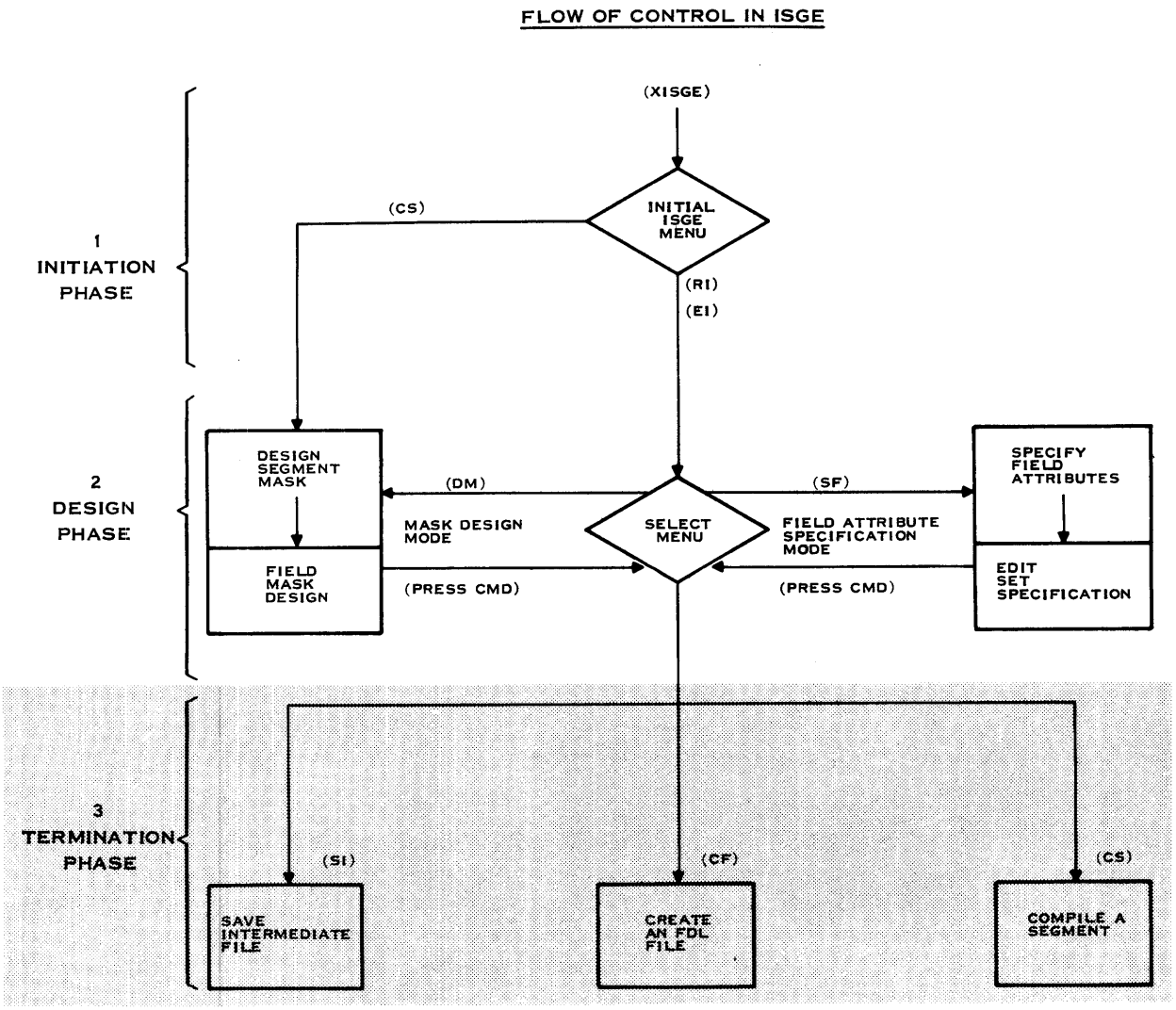


Figure 4-20. Flow of Control: Termination Phase

The cursor disappears from the screen while this process begins. The following message appears on your screen: FDL COMPILATION HAS BEGUN. Press **RETURN.** Enter the **WAIT** command. When the background execution is complete, proceed as follows:

Press **RETURN.**

To check for errors, enter the Show File (SF) command for the file .EXAMPLE.LIST. (Appendix C explains the FDL error count format.) If you followed directions, your compilation should be error free.

4.3.4.1 Save Intermediate File (SI). When you select this option, the ISGE prompts you for a pathname where the session's intermediate segment file can be saved.

4.3.4.2 Create an FDL File (CF). When you select this option, the ISGE prompts you for two pathnames. The first pathname is for a file where the session's intermediate segment file can be saved. This pathname is optional. The second pathname specifies a source file where the FDL translation can be stored. This pathname is required. ISGE translates the intermediate segment file into FDL statements, which are stored in the source file.

The CF option calls the FDL builder as a background task. Control immediately returns to SCI. The FDL builder task is in execution when SCI regains control. The FDL translation is not available until this background task terminates.

If you enter an invalid pathname for saving the intermediate segment file, the ISGE returns an error code. (See Appendix C.) You can either abort the session or enter another pathname. If you enter an invalid pathname for the FDL translation file, the ISGE terminates normally. The FDL builder detects the error and displays the COBOL error code. See Appendix C for techniques for recovering from this error.

4.3.4.3 Compile a Segment (CS). When you select this option, the ISGE prompts you for four file pathnames. The first pathname specifies a file where the intermediate segment file can be stored. This pathname is optional. The second file pathname specifies a file where the FDL translation can be stored. It is required. The third pathname specifies a program file into which the FDL is compiled. It is required. The fourth pathname specifies the FDL compilation listing file. It is required.

The CS option calls the FDL builder as a foreground task. Therefore, the terminal is unavailable until the FDL translation is complete. Once the translation to FDL is complete, the FDLC is bid as a background task. The terminal is returned to SCI after you press the RETURN key to acknowledge the message that FDL compilation has begun. The compiled segment is not available until the FDL compiler terminates.

If you specify an invalid pathname for the FDL source file, the ISGE terminates normally; the FDL builder detects the error and displays a COBOL error code. (See Appendix C for techniques for recovering from this kind of error.) If you specify an invalid pathname for either the program file or the listing file, the FDL compiler displays an error message. To recover from the error, execute the compiler by entering the command XFDLC and substitute a valid pathname for the pathname that was in error.

4.3.5 Form Tester

The Form Tester is a utility that allows the form designer to test a completed form without designing an application program. Section 7 provides a full description of the Form Tester. Test the segment created in this tutorial as follows:

Enter **FORMTSTR.** (The TIFORM Test Program menu appears on your screen.)
Enter **01** to open the form.

Note that the Form Tester accepts only default device-dependent segments. If a segment is device dependent, simply remove the DEVICE statement from the source and then test the segment. After testing, replace the appropriate DEVICE statement.

Your screen should now display the prompts for opening a form. Make the following entries:

Enter **.EXAMPLE.PROG** for PROGRAM FILE NAME.
Press **RETURN.**
Enter **ORDERF** for FORM NAME.
Accept the initial value of **ME** for TERMINAL NAME.
Press **RETURN.**
Enter **Y** for Sure.

The cursor is now in the bottom right-hand corner of your screen. Return to the TIFORM Test Program menu as follows:

Press **RETURN.**

Prepare the segment as follows:

Enter **02** for Prepare a Segment.
Press **RETURN.**

Your screen should now display the prompts for preparing a segment. Proceed as follows:

Enter **ORDERS** for SEGMENT NAME.
Press **RETURN.**
Enter **ORDERS** for READ GROUP.
Press **RETURN.**
Enter **Y** for SURE.

The form shown in Figure 4-11 appears on your screen. Enter sample data into each of the fields to test the attributes you specified in this tutorial. After you test the last field, proceed as follows:

Press **RETURN.**

A message appears in the middle of the screen, telling you what data was read. The cursor is in the bottom right corner of your screen. Return to the TIFORM Test Program menu as follows:

Press **RETURN.**

The TIFORM Test Program menu appears on your screen. Exit this program as follows:

Enter **22.**
Press **RETURN.**
Enter **Y** for SURE.
Press **CMD.**

4.3.6 Summary

This tutorial has introduced you to the basic features of ISGE. You created a segment mask, designed field masks, specified several commonly used field attributes, compiled your segment, and used the Form Tester. Although the form you created in this tutorial is relatively simple, ISGE can create very complex forms. See Appendix G for a list of all the attributes that can be assigned to the fields in a segment and a description of their functions. See Appendix D for additional examples of ISGE segments. Then you can modify the form you created in this tutorial by incorporating additional field attributes.

4.4 INTERMEDIATE SEGMENT FILE

The intermediate segment is the segment currently being operated on in an ISGE session. During execution of the ISGE, this segment is stored in a file called the intermediate segment file. Since the ISGE generates the intermediate segment file automatically, you need not be concerned with its creation or format.

You can save the intermediate segment file by using any of the three standard routines for terminating a session. (See Figure 4-20.) Upon selecting one of the three termination routes, you are prompted for a pathname under which the intermediate segment file can be saved. Specifying this pathname ensures that the intermediate segment file will be saved. You can then retrieve the intermediate segment file for editing during a subsequent ISGE session by selecting the Edit Intermediate Segment (EI) option from the initial ISGE menu (Figure 4-21).

A prompt will appear on your screen asking for a file pathname for the intermediate segment file. Specify the pathname under which the intermediate segment file you want to edit is stored. For example, to edit the intermediate segment file created during the tutorial in this section, specify .EXAMPLE.IMS. ISGE retrieves the file and the Selection menu appears. You can now proceed with the normal design and termination phases you completed in the tutorial.

4.5 CHANGING A COMPILED SEGMENT

You can change a compiled segment with the current version of TIFORM in either of two ways:

- To correct syntactical errors or context errors, use the Text Editor to change the appropriate FDL statements in the source file.

- To reposition fields or to redesign a segment mask or field mask, return to the ISGE and edit the intermediate segment file.

Note that in the current version of TIFORM you cannot edit a compiled segment by using the EC command from the initial ISGE menu (Figure 4-5). If you attempt to use this command to decompile a segment created with the current version of TIFORM, an error message appears.

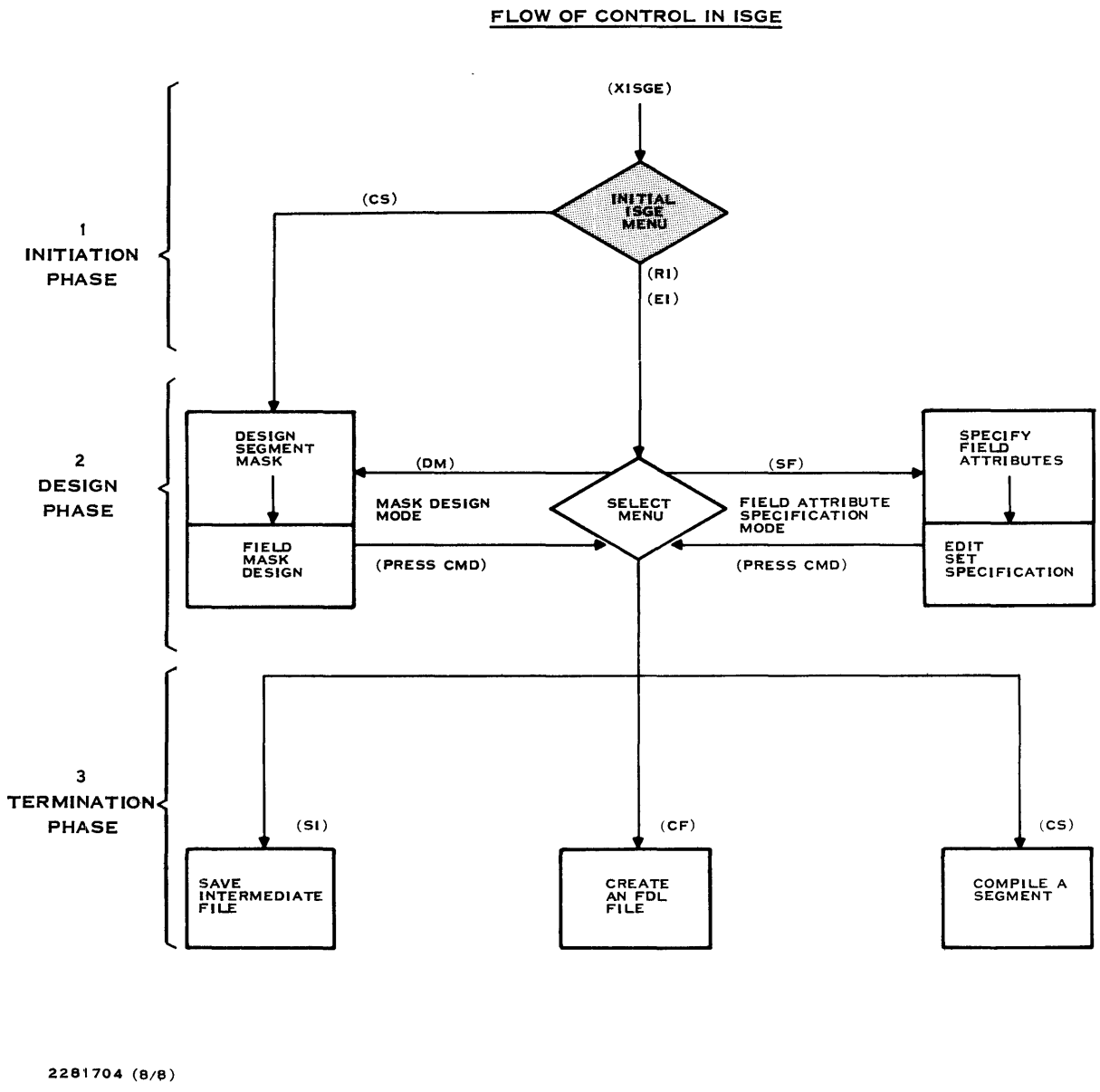


Figure 4-21. Flow of Control: Initial ISGE Menu

5

Application Interface

5.1 INTRODUCTION

The TIFORM application program interface routines provide access to the Form Executor. These routines reside in the TIFORM high-level language (HLL) interface packages, of which there are five. Through these packages, the application issues commands and passes data to the Executor and receives status and data in return. All intertask or interprocess communication between the application and the Executor is handled by these packages, thus providing consistent, error detecting interfaces. This section describes the calling sequences necessary to issue the various routines in the supported languages.

5.2 APPLICATION INTERFACE PACKAGES

The application interface to the Form Executor supports three application languages. These are COBOL, FORTRAN, and Pascal. COBOL is supported by two separate interface packages, one for the 3.1 and previous releases of COBOL and one for the 3.2 and later releases. Paragraph 5.2.1 discusses the COBOL interface packages.

Two interface packages are provided for Pascal. One package requires beginning and ending addresses of buffers. The other interface requires beginning addresses and *sizes* of buffers. Paragraph 5.2.2 discusses the Pascal interface packages.

One FORTRAN interface package is provided. You can use this package with FORTRAN-78 but not with FORTRAN IV (FORTRAN-66). Paragraph 5.2.3 discusses the FORTRAN interface package.

5.2.1 COBOL Application Interface

COBOL is supported by two separate interface packages, one for the 3.1 and previous releases of COBOL and one for the 3.2 and later releases. While you can use either the 3.2 or 3.1 calling sequences with COBOL 3.2, it is recommended that you use the calling sequence for 3.2 and later. You must use the 3.1 calling sequences with the 3.1 or earlier release of the COBOL language.

NOTE

Throughout this section, COBOL 3.1 refers to COBOL 3.1 and previous releases. COBOL 3.2 refers to COBOL 3.2 and later releases.

Some commands require parameters for the addresses of user data areas of variable size. The 3.2 COBOL interface makes use of COBOL run-time subroutines that provide access to the application data structure. The COBOL 3.1 interface cannot make use of these subroutines. Therefore, there are certain differences between the 3.2 and the 3.1 COBOL interfaces.

The major difference is that the 3.1 interface cannot tell implicitly the size of variable data areas. This requires that the 3.1 COBOL user pass both the beginning and ending addresses of variable data areas, while 3.2 COBOL users need only pass the beginning addresses. Variable data areas are those that have a size determined by the application. Variable data areas are generally buffers that are used for the exchange of data with the terminal.

For both 3.1 and 3.2 COBOL, data areas that are not of variable size are identified by passing only the beginning address of the areas. Fixed-length data areas include segment names, group names, index fields, and count fields.

The following examples illustrate the use of these parameter types.

EXAMPLE 1

```
*
*  3.1 COBOL CALLING SEQUENCE EXAMPLE
************************************
01 GROUP1-NAME PIC X(6) VALUE 'GROUP1'.
01 DATA1-BEGIN PIC X(100).
01 DATA1-END   PIC X.
      .
      .
   CALL "CF$REA" USING GROUP1-NAME, DATA1-BEGIN, DATA1-END.
```

EXAMPLE 2

```
*
*  3.2 COBOL CALLING SEQUENCE EXAMPLE
************************************
 01 GROUP-TABLE.
    03 GROUP1-NAME PIC X(6) VALUE 'GROUP1'.
    03 GROUP2-NAME PIC X(6).
       .
       .
 01 DATA-BUFFER-AREA.
    03 DATA1 PIC X(45).
    03 DATA2 PIC X(20).
       .
       .
    CALL "CX$REA" USING GROUP1-NAME, DATA1.
       .
```

In the 3.1 COBOL example, the group name in item GROUP1-NAME is fixed length, six bytes. This parameter is always the same length in every CF$REA call. The read data area, represented by DATA-BEGIN and DATA-END, may vary in size in different CF$REA calls. Consequently, you must specify both the start and end addresses so that CF$REA can know how large a buffer it has to contain the data it reads.

In the 3.2 COBOL example, the group name in GROUP1-NAME is also fixed at six bytes. The read data area is represented by DATA1 and the size of DATA1 — PIC X(45). An explicit item to indicate the end of the data area is not required.

The term *group* is used here for all the Read, Write, and Reset commands, but it is always implied that you can use an external field or variable name instead. Indexing is meaningful for groups only; otherwise, you can use a field name or a variable name anywhere that a group name is specified.

The previous example points out another difference. With 3.1 or earlier COBOL, all items used as parameters must be either 01-level or 77-level items. That is, they must start on word boundaries.

These restrictions do not apply with the use of 3.2 or later COBOL. As the example illustrates, all items used as parameters can be at any level and there is no requirement for starting on word boundaries. The only exception is that the status block declared by the CX$STS call must be on a word boundary, so you must declare it at the 01 level.

For both the 3.1 and 3.2 interfaces, form names, segment names, group names, field names, and variable names are assumed to reside left-justified in a six-character data item, blank-filled on the right. Index and count items are assumed to reside right-justified in a three-character data item, zero-filled on the left.

5.2.1.1 COBOL 3.2 Calling Sequences. The entry points provided for COBOL 3.2 access to the application interface routines are all of the form CX$xxx. Table 5-1 lists these names. The syntax of the call to each of these entry points is included in the paragraph that describes the functions performed by each call.

Table 5-1. COBOL 3.2 Entry Points to Application Interface Routines

Calls	Meaning
CX$STS	Declare Status Block
CX$OF	Open Form
CX$PS	Prepare Segment
CX$WRI	Write a Group
CX$REA	Read a Group
CX$WWR	Write With Reply
CX$WX	Write Indexed
CX$REX	Read Indexed
CX$RXC	Read Indexed With Cursor Return
CX$WXR	Write Indexed With Reply
CX$WRC	Write Indexed With Reply and Cursor Return
CX$WM	Write Message
CX$AEK	Arm Event Keys
CX$DAK	Disarm Event Keys
CX$CN	Control Functions
CX$RF	Reset Form
CX$RFX	Reset Form Indexed
CX$PKY	Print Key Command
CX$CHF	Change Form
CX$ASN	Execute Asynchronously
CX$SYN	Synchronize
CX$CIC	Change ITC/IPC Communication
CX$CF	Close Form

5.2.1.2 **COBOL 3.1 Calling Sequences.** You can only use the COBOL 3.1 calling sequences with 3.1 and earlier releases of COBOL. The entry points provided for COBOL 3.1 (and earlier releases) access to the application interface are of the form CF$xxx, where xxx is the same as for the entry points listed in Table 5-1. The syntax of the call to each entry point is included in the paragraph that discusses the functions performed by each call. An example of the use of each call is also included.

5.2.2 Pascal Application Interface

Pascal is supported by two interface packages. The first interface package requires beginning and ending addresses of variable-length buffers. The second interface package requires the use of beginning addresses and *sizes* of variable-length buffers.

Entry points for the first package (type 1) are of the form PF$aaa, where aaa defines the unique entry point. Entry points for the second package (type 2) are of the form PX$aaa.

The type 1 package uses the EXTERNAL FORTRAN interface, while the type 2 package uses the more efficient EXTERNAL interface and the Pascal upper bound (UB< array>) function.

Any release of Pascal supports either interface package. However, since the packages are linked separately, you cannot mix calls to the entry points.

Table 5-2 lists the entry points of the package requiring beginning addresses and sizes (type 2). The module .PASCAL.PX$START on the object installation media contains the correct Pascal definitions for each of these entry points.

For the type 1 interface package, you must declare each entry point used by the Pascal program as EXTERNAL FORTRAN. You must declare each parameter as a variable (VAR).

For type 2, each entry point must be EXTERNAL. You must declare each parameter as a variable parameter with the exception of all six-character names (for example, form names and group names) and parameters that represent the size of data areas. You must not declare these parameters as variable. In addition, you must define all data areas used with the type 2 interface as dynamic arrays. The lower bound of all such arrays must be one. An example follows.

```
VAR READ_DATA:PACKED ARRAY[1..?] OF CHAR;
```

It is recommended that the Pascal upper bound (UB< array>) be used for determining the size of each array.

Even though the syntax of some entries indicates optional parameters, you must always use the same number of optional entries to avoid a Pascal error.

Table 5-2. Pascal Entry Points to Application Interface Routines

Calls	Meaning
PX$STS	Declare Status Block
PX$OF	Open Form
PX$PS	Prepare Segment
PX$WRI	Write a Group
PX$REA	Read a Group
PX$WWR	Write With Reply
PX$WX	Write Indexed
PX$REX	Read Indexed
PX$RXC	Read Indexed With Cursor Return
PX$WXR	Write Indexed With Reply
PX$WRC	Write Indexed With Reply and Cursor Return
PX$WM	Write Message
PX$AEK	Arm Event Keys
PX$DAK	Disarm Event Keys
PX$CN	Control Functions
PX$RF	Reset Form
PX$RFX	Reset Form Indexed
PX$PKY	Print Key Command
PX$CHF	Change Form
PX$ASN	Execute Asynchronously
PX$SYN	Synchronize
PX$CIC	Change ITC/IPC Communication
PX$CF	Close Form

Note:

Type 2 requires beginning addresses and sizes.

The following example shows the use of the type 1 application interface package, which requires beginning and ending addresses. The procedure declarations use EXTERNAL FORTRAN.

EXAMPLE

```
PROGRAM FORMS_DEMO; { TIFORM example in TI Pascal }
                    {Type 1: end addresses are required.}
TYPE
    C$2 = PACKED ARRAY[1..2] OF CHAR;

    C$6 = PACKED ARRAY[1..6] OF CHAR;

    STATUS_BLOCK = PACKED RECORD
       FORM_STATUS:C$2;
       OPSYS_STATUS:C$2;
       EVENT_KEY:C$2;
       COMMAND: C$2;
       ITEM_NAME: C$6;
```

```
      {INDEX_01,  INDEX_02,ITEM_CNT,CURSOR_POSITION are
       allocated as one 12-character array since arrays are
       allocated on word boundaries. Generally, you can
       ignore these fields.}
    INDEXES_CURSOR : PACKED ARRAY[1..12] OF CHAR;
       TEXT_LENGTH: INTEGER;
       FILLER_1: PACKED ARRAY[1..12] OF CHAR;
    END;

      {DIR_NAME may vary from 2-48 bytes depending on
       expected size of pathnames for the
       form program file}
    DIRECT_NAME = RECORD
       DIR_NAME:PACKED ARRAY[1..48] OF CHAR;
       D_N_END:INTEGER
    END;

      {TRM_NAME may vary from 2-8 bytes depending on the
       expected size of the device name or synonym to be
       used for the terminal.}
    TERMINAL_NAME = RECORD
       TRM_NAME:PACKED ARRAY[1..4] OF CHAR;
       T_N_END:INTEGER
    END;

      {Vary R_BUFF to accommodate maximum user input for any
       group read.}
    READ_BUFFER = RECORD
       R_BUFF:PACKED ARRAY[1..40] OF CHAR;
       R_B_END:INTEGER
    END;

VAR SBLOCK: STATUS_BLOCK;
    D_NAME: DIRECT_NAME; T_NAME: TERMINAL_NAME;
    R_BUFFER: READ_BUFFER;
    GROUP,SEGMENT,FORM:C$6;

PROCEDURE PF$STS (VAR SBLK:STATUS_BLOCK); EXTERNAL FORTRAN;

PROCEDURE PF$OF  (VAR FNAME:C$6;
                  VAR DIRNME:DIRECT_NAME;
                  VAR DIREND:INTEGER;
                  VAR TRMNME:TERMINAL_NAME;
                  VAR TRMEND:INTEGER); EXTERNAL FORTRAN;

PROCEDURE PF$PS  (VAR SNAME: C$6); EXTERNAL FORTRAN;

PROCEDURE PF$REA (VAR GNAME:C$6;
                  VAR RDDATA:READ_BUFFER;
                  VAR RDDEND:INTEGER); EXTERNAL FORTRAN;

PROCEDURE PF$CF; EXTERNAL FORTRAN;
```

```
BEGIN
   WITH D_NAME,T_NAME,R_BUFFER DO
   BEGIN
      PF$STS   (SBLOCK); {Declare the status block.}

      DIR_NAME[1]:=' ';
            {Use default synonym DIRECTRY for form file.}
      TRM_NAME:='TERM';  {Use synonym TERM for terminal.}
      {Open form.}
      PF$OF (FORM, D_NAME, D_N_END, T_NAME, T_N_END);


      SEGMENT:='SEG1  ';
      PF$PS (SEGMENT); {Activate the segment.}

      GROUP:='GROUP1';
      PF$REA (GROUP,R_BUFFER,R_B_END); {Read group.}

                     {Process user input.}

      PF$CF
   END; {Close the form.}
END.
```

The following example shows the same program. However, it is coded to demonstrate the type 2 application interface package. Note that the Pascal function UB is used to provide the size of variable length areas and that these size parameters are not variable.

EXAMPLE

```
PROGRAM FORMS_DEMO; { TIFORM example in TI PASCAL }
                    {Type 2: sizes are required.}
TYPE
    C$2 = PACKED ARRAY[1..2] OF CHAR;
    C$6 = PACKED ARRAY[1..6] OF CHAR;
    STATUS_BLOCK = PACKED RECORD
       FORM_STATUS:C$2;
       OPSYS_STATUS:C$2;
       EVENT_KEY:C$2;
       COMMAND: C$2;
       ITEM_NAME: C$6;

    INDEXES_CURSOR : PACKED ARRAY[1..12] OF CHAR;
       TEXT_LENGTH: INTEGER;
       FILLER_1: PACKED ARRAY[1..12] OF CHAR;
    END;
      {INDEX_01,  INDEX_02,  ITEM_CNT,  CURSOR_POSITION are
       allocated as one 12-character array since arrays are
       allocated on word boundaries. The user may ignore
       these fields.}
```

```
    DIR_NAME=PACKED ARRAY[1..10] OF CHAR;
      {DIR_NAME may vary from 2-48 bytes depending on
       expected size of pathnames for the
       form program file.}

    TRM_NAME=PACKED ARRAY[1..4] OF CHAR;
      {TRM_NAME may vary from 2-8 bytes depending on the
       expected size of the device name or synonym to be
       used for the terminal.}

    R_BUFF=PACKED ARRAY[1..48] OF CHAR;
      {Vary R_BUFF to accommodate maximum user input for any
       group read.}
VAR SBLOCK: STATUS_BLOCK;
    D_NAME: DIR_NAME; T_NAME: TRM_NAME;
    R_BUFFER: R_BUFF;
    GROUP,FORM:C$6;

PROCEDURE PX$STS (VAR SBLK:STATUS_BLOCK); EXTERNAL;

PROCEDURE PX$OF  (FNAME:C$6;
                  VAR DIRNME:PACKED ARRAY[1..?] OF CHAR;
                    DIRSIZ:INTEGER;
                  VAR TRMNME:PACKED ARRAY[1..?] OF CHAR;
                    TRMSIZ:INTEGER);
                  EXTERNAL;

PROCEDURE PX$PS  (SNAME:C$6);
                   EXTERNAL;

PROCEDURE PX$REA (GNAME:C$6;
                   VAR RDDATA:PACKED ARRAY[1..?] OF CHAR;
                       RBFSIZ:INTEGER);
                   EXTERNAL;

PROCEDURE PX$CF;
                   EXTERNAL;

BEGIN
       PX$STS   (SBLOCK); {Declare the status block.}

       D_NAME:=' ';{Use default synonym DIRECTRY for
              form file.}
       T_NAME:='TERM';  {Use synonym TERM for terminal.}

       FORM:='FORM01';  {Actual form name.}
       PX$OF (FORM, D_NAME, UB(D_NAME), T_NAME,
              UB(T_NAME)); {Open the form.}
```

```
        PX$PS ('SEG001'); {Literal segment name is used.}

                {Activate the segment.}

        GROUP:='GROUP1';
        PX$REA (GROUP,R_BUFFER,UB(R_BUFFER)); {Read user input.}

               {Process user input.}

        PX$CF   {Close the form and terminate the Executor.}
END.
```

In the following paragraphs, only the type 2 syntax and examples are presented. For these examples, the following types are understood:

```
C$2 = PACKED ARRAY[1..2] OF CHAR;
C$3 = PACKED ARRAY[1..3] OF CHAR;
C$6 = PACKED ARRAY[1..6] OF CHAR;
```

You can derive the syntax of type 1 calls from the type 2 syntax by the following method:

- Changing the call from PX$aaa to PF$aaa.
- Replacing < iiii size> by < iiii end address>, where iiii is the name of a variable-sized item (pathname, buffer, and so on).
- Making all parameters VAR.

In addition, < iiii > (beginning of iiii) and < iiii end address> must be in a record to ensure that the size of iiii is computed correctly by the interface package.

The parameters referred to as < index-1>, < index-2>, < count>, and < cursor position> must be specified as three-character packed arrays. The values of these parameters are represented as character strings. They are not numeric integers upon which arithmetic can be performed. ENCODE and DECODE can be used for conversion.

Each type 1 procedure declaration must be EXTERNAL FORTRAN rather than the type 2 EXTERNAL declaration.

5.2.3 FORTRAN Application Interface

FORTRAN applications are supported by a single application interface package. You can use the interface package for an application written in FORTRAN-78 but not FORTRAN IV (FORTRAN-66). Entry points are of the form FFaaa, where aaa defines the unique entry point for each function. Table 5-3 lists these entry points. The aaa of each entry point corresponds to entry points in the COBOL CF$ and the Pascal PX$ application interface package, which perform the identical functions.

Table 5-3. FORTRAN Entry Points to Application Interface Routines

Call	Meaning
FFSTS	Declare Status Block
FFOF	Open Form
FFPS	Prepare Segment
FFWRI	Write a Group
FFREA	Read a Group
FFWWR	Write With Reply
FFWX	Write Indexed
FFREX	Read Indexed
FFRXC	Read Indexed With Cursor Return
FFWXR	Write Indexed With Reply
FFWRC	Write Indexed With Reply and Cursor Return
FFWM	Write Message
FFAEK	Arm Event Keys
FFDAK	Disarm Event Keys
FFCN	Control Functions
FFRF	Reset Form
FFRFX	Reset Form Indexed
FFPKY	Print Key Command
FFCHF	Change Form
FFASN	Execute Asynchronously
FFSYN	Synchronize
FFCIC	Change ITC/IPC Communication
FFCF	Close Form

The following example shows the use of TIFORM in a FORTRAN-78 program.

EXAMPLE

```
            .
            .
      CHARACTER*2   STATUS(20),BUFFER(11),NINE
      CHARACTER*2   DRCTRY(2), TRMNL(2), INBUF(21)
      CHARACTER*3   INDEX1, INDEX2, ECOUNT, CURPOS
C
      DATA  NINE/'09'/, INDEX1/'000'/, INDEX2/'000'/
      DATA  ECOUNT/'000'/, CURPOS/'000'/
            .
            .
C
C     Declare the status block...
C
      CALL  FFSTS (STATUS)
      DRCTRY(1) = '  '
      TRMNL(1) = '  '
C
C     Open the form...
C
```

```
      CALL  FFOF ('FORM1 ', DRCTRY(1), DRCTRY(2),
     1                       TRMNL(1),  TRMNL(2))
C
      IF  (STATUS(1) .GT. NINE)   GO TO  1000
C
C     Prepare a segment...
C
      CALL  FFPS ('SEGM1 ')
C
      IF  (STATUS(1) .GT. NINE)   GO TO  1000
C
C     Read 40 bytes of the entire segment...
C
      CALL  FFRXC ('SEGM1 ', INBUF, INBUF(21),
     1               INDEX1, INDEX2, ECOUNT, CURPOS)
C
      IF  (STATUS(1) .GT. NINE)   GO TO  1000

      Process the data

C
C     Close the form...
C
 1000 CALL FFCF
             .
             .
      END
```

5.3 INDEXED OPERATIONS

Some TIFORM commands allow the application program to specify four indexing parameters. These parameters (in order) are a first-level index, a second-level index, an elemental member count, and an intrafield cursor position. A command in which these parameters are legal is called an indexed command.

The following example shows an indexed read command expressed in COBOL and using the COBOL 3.1 interface.

EXAMPLE

```
          .
          .
01  GROUP-NAME  PIC X(6) VALUE IS "NAMEXX".
01  DATA-BEGIN  PIC X(80).
01  DATA-END    PIC XX.
01  INDEX-1     PIC 999.
01  INDEX-2     PIC 999.
01  COUNT       PIC 999.
01  CURSOR-POS  PIC 999.
          .
          .
```

```
MOVE 03 TO INDEX-1.
MOVE 09 TO INDEX-2.
MOVE 06 TO COUNT, CURSOR-POS.
CALL "CF$REX" USING GROUP-NAME,
                    DATA-BEGIN, DATA-END,
                    INDEX-1, INDEX-2, COUNT,
                    CURSOR-POS.
```

When these indexing parameters are allowed on a call, they are optional. The application can specify as many of these parameters on a given call as necessary to perform the desired operation. Since the parameters' identities within the call are determined positionally, if one is omitted, all following indexing parameters must also be omitted. An omitted indexing parameter is assumed to have a value of zero.

You can specify the indexing parameters to condition the basic command being performed. The indexing parameters can also have values returned in them by the Form Executor. The following paragraphs discuss the meanings of the indexing parameters in both situations.

The first and second indexing parameters are actually indexes into the structure of the group specified in the command. The first index identifies a member of the named group. If the member selected by the first index is itself a group, you can specify the second index to identify a member of the group selected by the first index.

An indexed command always starts with the first elemental member of the group/field/variable selected by the indexing, where the first (and only) elemental member of a field/variable is defined to be the field/variable itself.

When you specify cursor return in a Read or a Write With Reply command and an armed event key or the Enter key is pressed, these two indexing parameters are modified by the command to point to the field containing the cursor. If the application read is terminated by any other means, these two parameters are set to zero since the cursor position is indeterminate.

The third indexing parameter is called the elemental member count parameter, or count. It specifies the number of elemental members of the named group to be read, written, or reset by the command, starting with the elemental member selected by the indexes. If the count is zero on a Write or a Reset command, the remainder of the named group's elemental members are written or reset. If the count is zero on a Read command, the entire named group is read, but the cursor is initially placed on the elemental member selected by the indexing. Since a Write With Reply command is in effect a write followed by a read, a zero count on an Indexed Write with Reply command has two quite different effects and should be avoided.

When you specify cursor return in a Read or a Write with Reply command and the command is terminated by an armed event key or the Enter key, the elemental member count parameter is set to the elemental number of the field containing the cursor. A field's elemental number is equal to one plus the number of elemental members preceding it in the command's specified group. If the Read command is terminated in any other way, the count parameter is set to zero since the cursor position is indeterminate.

The fourth indexing parameter is called the intrafield cursor position, or cursor. This parameter specifies the character position within the indexed field where the cursor is placed initially. It is used by Read commands only, although you can specify it on any indexed command. If you omit it or specify a zero value, the cursor is positioned in the first character position of the indexed field.

If you specify cursor return and the Read command is terminated by an armed event key, the position of the cursor within the current field is returned in the cursor parameter. If the Read is closed with the Enter key, the value returned is the length of the field plus one. If the Read is closed in any other way, a zero value is returned since the cursor position is indeterminate.

5.4 STATUS BLOCK

All TIFORM commands use a 40-byte status block to return common information about the command most recently executed. The area to be used is declared to TIFORM in a Declare Status Block call. (Refer to the description of the Declare Status Block routine.) Subsequent commands use the area specified on the most recent Declare Status Block call. Only the first six bytes of the status block are generally useful. The following example shows the format of the status block, coded in COBOL.

EXAMPLE

```
*
*    FIELDS WITHIN TIFORM'S STATUS BLOCK...
*
 01  TIFORM-STATUS-BLOCK.
     05  FORM-STATUS              PIC 9(02).
     05  OPSYS-STATUS             PIC X(02).
     05  EVENT-KEY                PIC 9(02).
     05  COMMAND                  PIC X(02).
     05  ITEM-NAME                PIC X(06).
     05  INDEX-01                 PIC 9(03).
     05  INDEX-02                 PIC 9(03).
     05  ITEM-CNT                 PIC 9(03).
     05  CURSOR-POSITION          PIC 9(03).
     05  TEXT-LENGTH              PIC 9(05) USAGE IS COMP-1.
     05  FILLER                   PIC X(12).
```

The FORM-STATUS field's value indicates whether the Form Executor was able to completely execute the last command. A zero value indicates a successful execution. A nonzero value indicates some kind of exceptional condition. A value greater than nine indicates a fatal error. A value less than or equal to nine indicates that a special, nonfatal condition occurred on the command. Appendix B lists the possible form status codes and their meanings.

NOTE

Your program should check the FORM-STATUS field after every command.

The OPSYS-STATUS field contains the hexadecimal operating system status code if an SVC caused the error; otherwise, this field contains zeros. This field is intended for reporting purposes only. Not all form status codes generate an SVC status; however, for those that do, there is always only a single SVC that generates the SVC status. Appendix B describes the association between form status codes and SVCs.

The EVENT-KEY field contains the function key code of the event key that terminated the last command. This value is zero unless an armed function key or a special event key terminated the last command. Refer to Table 2-2 or 2-4 for a list of the codes for the function keys. This field is meaningful only when the previous command was one of the Read commands.

The remainder of the fields in the status block are filled in by the interface routines (to communicate information to the Form Executor) and in turn are filled in and returned by the Form Executor. It is from these status block fields that routines like CF$RXC obtain index and count values from the Executor. After a command executes, these fields are usually all zeros. The sole exception occurs on the Read commands, when the cursor position is returned in them.

The TEXT-LENGTH field contains the number of bytes of text (exclusive of the status block) that were transmitted back to the application by the Executor. If more text than the application's buffer can hold is sent back by the Executor, this field indicates how much data actually is sent. This field is meaningful only after Read and Write with Reply commands.

The application program is not required to initialize the status block prior to a call. The interface routines set all statuses to zeros at the start of each command.

5.5 ITEMS RETURNED FROM READ COMMANDS

The Read commands, in addition to the status codes and their input data, return several other pieces of information. All Read commands return an event key code in the status block. If the Read command was terminated by an armed event key, the value returned is one of the codes shown in Table 2-2 or 2-4. Otherwise, a value of zero is returned.

There are some special nonfatal form status codes that are returned only by Read or Write With Reply commands. They indicate the method by which the terminal user terminated the read. Table 5-4 lists these special codes.

If you specify cursor return on a Read or Write with Reply command and the user terminates the Read with an armed event key, the position of the cursor is returned in the indexing variables. If the user terminates the read with the Enter key, the first three indexing variables are returned and the cursor position is meaningless.

Table 5-4. Nonfatal Form Status Codes

Code	Meaning
00	Indicates that the user terminated the Read by pressing Next Field, Forward Tab, Skip, or Return in the last field of the read.
01	Indicates that the user terminated the Read by pressing Previous Field or Back Tab from the first field of the Read.
02	Indicates that the user terminated the Read by pressing an armed nonabort event key or the Enter key.
03	Indicates that the user terminated the Read by pressing an armed abort key.
06	Indicates that the read was terminated by a TERMINATE READ or a TERMINATE READ IMMEDIATELY statement.

5.6 ARM EVENT KEYS ROUTINE

This routine issues a command to arm event keys selected by the application. Function keys are the only event keys that can be armed by the application. The function keys specified by this call completely replace whatever function keys were armed prior to the call.

To specify that a function key is to be armed requires a three-byte string. The first two bytes are the function key code. The third byte is the specified function key's attributes. Table 2-2 or 2-4 lists the function key codes. The attribute codes are the character A for an abort key and a blank space for a nonabort key. To arm several function keys, the application must concatenate three-byte strings for each key, then pass the total string to the Form Executor via this command.

The effects of the use of armed keys are as follows. If the user presses a nonabort key, the current Read is terminated, all fields are verified and processed, and the data is returned to the application with a form status of 02. If the user presses an abort key, the current Read is terminated and its data is returned to the application with a form status of 03 and without any editing or processing. It is the application's responsibility to check the EVENT-KEY field of the status block after a Read command, to detect that an abort key terminated the Read, and to handle the abort key data. Data returned from a field that the user terminates with an abort key is frequently invalid or unusable. All function keys are disarmed by a Prepare Segment command.

5.6.1 Arm Event Keys Calling Sequences

The calling sequences for the Arm Event Keys routine are as follows:

COBOL 3.1:

```
CALL "CF$AEK" USING < event code list> , < event code list end> .
```

COBOL 3.2:

```
CALL "CX$AEK" USING < event code list> .
```

Pascal:

PX$AEK (< event code list> , < event code list size);

FORTRAN:

CALL FFAEK (< event code list> , < event code list end>)

5.6.2 Arm Event Keys Parameters

The following list describes the parameters for the Arm Event Keys routine:

<event code list> is the item (01-level item in COBOL 3.1) containing a string of three-byte function key specifications, left-justified, and either blank-filled or low-value (binary zero) filled on the right.

<event code list end> is the item (01-level item in COBOL 3.1) denoting the end of the function code list area.

<event code list size> is the item in Pascal indicating the size of the function code list area.

5.6.3 Arm Event Keys Results

Status is posted. The specified function keys are armed.

5.6.4 Arm Event Keys Examples

The following examples demonstrate the Arm Event Keys routine:

COBOL 3.1:

```
          .
          .
          .
 01  EV-CO-LST        PIC X(20) VALUE IS "SPACES".
 01  EV-CO-LST-END    PIC XX.
          .
          .
          .
* ENABLE F1, F2, F4, F5, F8, CMD
*     WITH F4 AND F8 ABORT KEYS
*
     MOVE "01 02 04A05 08A40" TO EV-CO-LST.
     CALL "CF$AEK" USING EV-CO-LST, EV-CO-LST-END.
```

Pascal:

```
            .
            .
            .
{Use CKEYS to arm the following keys:
{       F1 as an abort key
{       F2 as a nonabort key
{       F3 as an abort key
{ and   F4 as an abort key
```

```
CONST CKEYS = '01A02 03A04A';
              .
              .
              .
VAR EKEYS:PACKED ARRAY[1..12] OF CHAR;
              .
              .
              .
PROCEDURE PX$AEK (VAR KEYS:PACKED ARRAY[1..?] OF CHAR;
                  K_SIZE:INTEGER); EXTERNAL;
              .
              .
              .
EKEYS:=CKEYS;
PX$AEK (EKEYS,UB(EKEYS));
```

FORTRAN:

```
              .
              .
              .
 CHARACTER EKEYS*2(6)
 CHARACTER*3 EKEND
              .
              .
              .
 DATA EKEYS/'01A02 03A04A'/
C Use CKEYS to arm the following keys:
C        F1 as an abort key
C        F2 as a nonabort key
C        F3 as an abort key
C  and   F4 as an abort key
 CALL FFAEK (EKEYS(1),EKEYS(6))
```

5.6.5 Arm Event Keys Program Notes

Your application should check the status block after each Arm Event Keys command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.7 CHANGE FORM ROUTINE

This routine combines the functions of a Close Form routine followed by an Open Form routine. It is more flexible than these two routines, since any, all, or none of the form, the form program file, and the terminal can be changed on a single routine. Furthermore, this routine does not force the termination of one Form Executor and the bidding of a second Executor as does a Close/Open routine pair. Also, some of its options can suppress logical unit number (LUNO) assigns and releases. Change Form is therefore appreciably faster than a Close Form followed by an Open Form.

5.7.1 Change Form Calling Sequences

The calling sequences for the Change Form routine are as follows:

COBOL 3.1:

```
CALL "CF$CHF" USING <form name>, <directory name>, <directory name end>,
                    <terminal name>, <terminal name end>.
```

COBOL 3.2:

```
CALL "CX$CHF" USING <form name>, <directory name>, <terminal name>.
```

Pascal:

```
PX$CHF (<form name>, <directory name>, <directory name size>,
        <terminal name>, <terminal name size>);
```

FORTRAN:

```
      CALL FFCHF (<form name>, <directory name>, <directory name end>,
     1            <terminal name>, <terminal name end>)
```

5.7.2 Change Form Parameters

The following list describes the parameters for the Change Form routine:

<form name> can be any one of the following:

- A six-byte, 01-level item in COBOL 3.1, a six-character packed array in Pascal, or a six-element character array in FORTRAN containing the name of the form to be opened. This name must be left-justified and blank-filled on the right within this item.

- A 01-level, signed, COMP-1 item in COBOL 3.1 or an integer in Pascal and FORTRAN containing the value −1, indicating that the form is not to change. The effect of this option is that the overlay number of the form root of the currently open form is used to load the new form root during the Open Form phase of the command.

- A 01-level, signed, COMP-1 item in COBOL 3.1 or an integer in Pascal and FORTRAN containing the overlay number of the form root of the new form that is to be opened during the Open Form phase of the command.

< directory name> can be any one of the following:

- The starting address of the 2-48 byte item in COBOL 3.1, a 2-48 character packed array in Pascal, or a 2-48 element character array in FORTRAN containing the operating system pathname specifying the program file in which the specified form is stored. This item's value must be left-justified and blank-filled to the right. Synonyms are allowed. If the first type is a blank, the synonym DIRECTORY is assumed to point to the program file.

- A 01-level, signed, COMP-1 item in COBOL 3.1 or an integer in Pascal and FORTRAN containing the value – 1, indicating that the form program file is not to change. This option suppresses the release and assign of a LUNO to the form program file. Instead, the LUNO assigned to the previously active form program file is used when loading the form root of the form being opened by this command.

- A 01-level, signed, COMP-1 item in COBOL 3.1 or an integer in Pascal and FORTRAN containing the LUNO of the new form program file in which the new form is to be found. This option suppresses the assign of a LUNO to the form program file. Instead, the LUNO specified is used when loading the form root of the form being opened by this command. The LUNO must have been previously assigned to the form program file and must be either a station local or global LUNO.

< directory name end> in COBOL 3.1 and FORTRAN must always represent the end address of the < directory name> parameter. In particular, if < directory name> is a 01-level, signed, COMP-1 item, then < directory name end> must reside immediately following it. The difference between the < directory name> and < directory name end> parameters must be two bytes.

< directory name size> in Pascal must always represent the size of the < directory name> parameter. In particular, if < directory name> is integer, then < directory name size> must be two.

< terminal name> can be any one of the following:

- The start address of a 2-48 byte area in COBOL 3.1, the 2-48 byte packed character array in Pascal, or the 2-48 element character array in FORTRAN containing the device name of the terminal or the pathname of the file on which the form is to execute. The value of this item must be left-justified and blank-filled on the right. Synonyms are allowed. If the first byte of the area is a blank, the synonym ME (pointing to the application's controlling station) is used, causing the form to be executed on the application's own station.

- A 01-level, signed, COMP-1 item in COBOL 3.1 or an integer in Pascal and FORTRAN containing the value – 1, indicating that the terminal to be used is not to change. This option suppresses the release and assign/open of a LUNO to the terminal. Instead, the LUNO already assigned to the currently active terminal is used for all subsequent terminal I/O.

- A 01-level, signed, COMP-1 item in COBOL 3.1 or an integer in Pascal and FORTRAN containing the LUNO assigned to the new terminal on which the specified form is to be executed. This option suppresses the assign of a LUNO to the terminal. Instead, the specified LUNO is opened and used for all subsequent terminal I/O.

< terminal name end> in COBOL 3.1 and FORTRAN must always represent the end of the < terminal name> parameter. In particular, if < terminal name> is a 01-level, signed, COMP-1 item, then < terminal name end> must reside immediately following it. The difference between < terminal name> and < terminal name end> must be two bytes.

< terminal name size> in Pascal must always represent the size of the < terminal name> parameter. In particular, if < terminal name> is an integer, then < terminal name size> must be two.

5.7.3 Change Form Results

Status is posted. The currently open form is closed, and the form specified by the call is opened on the specified terminal.

5.7.4 Change Form Examples

The following examples demonstrate the Change Form routine:

COBOL 3.1 Example 1:

```
              .
              .
              .
* A STANDARD CHANGE FORM CALL.
* ****************************
*
 01 FORM-NAME       PIC X(6) VALUE IS "FORM01".
 01 DIR-NAME        PIC X(8) VALUE "PATHNAME".
 01 DIR-NAME-END    PIC XX.
 01 TERM-NAME       PIC XX VALUE "ME".
 01 TERM-NAME-END   PIC XX.
              .
              .
              .
    CALL "CF$CHF" USING FORM-NAME,
                        DIR-NAME, DIR-NAME-END,
                        TERM-NAME, TERM-NAME-END.
```

COBOL 3.1 Example 2:

```
              .
              .
              .
* CHANGE ONLY THE FORM.  DIRECTORY, TERMINAL ARE UNCHANGED.
*************************************************************
*
 01 FORM            PIC X(6) VALUE IS "FORM02".
 01 DIR             PIC S9(5) USAGE IS COMP-1 VALUE -1.
 01 DIR-END         PIC XX.
 01 TERM            PIC S9(5) USAGE IS COMP-1 VALUE -1.
 01 TERM-END        PIC XX.
              .
              .
              .
    CALL "CF$CHF" USING FORM,
                        DIR, DIR-END,
                        TERM, TERM-END.
```

Pascal Example 1:

```
{ A STANDARD CHANGE FORM CALL.
 ============================}

            .
            .
            .
CONST C_FORM = 'FORM01';
      C_DIRC = 'PATHNAME';
      C_TERM = 'ST10';
            .
            .
            .
VAR FORM:C$6;
    DIRC:PACKED ARRAY[1..8] OF CHAR;
    TERM:PACKED ARRAY[1..4] OF CHAR;
            .
            .
            .
PROCEDURE PX$CHF (F_NAME:C$6;
                  VAR F_DIRC:PACKED ARRAY[1..?] OF CHAR;
                      F_DSIZ:INTEGER:
                  VAR F_TERM:PACKED ARRAY[1..?] OF CHAR;
                      F_TSIZ:INTEGER);
                  EXTERNAL;
            .
            .
            .
FORM:=C_FORM; DIRC:=C_DIRC TERM:=C_TERM;
PX$CHF (FORM,DIRC,UB(DIRC),TERM,UB(TERM));
```

Pascal Example 2:

```
    {CHANGE ONLY THE FORM.  DIRECTORY, TERMINAL ARE UNCHANGED.
    ========================================================}

              .
              .
              .
   VAR FORM:C$6;
       DIRC:PACKED ARRAY[1..8] OF CHAR;
       TERM:PACKED ARRAY[1..4] OF CHAR;
              .
              .
              .
   PROCEDURE PX$CHF (F_NAME:C$6;
                     VAR F_DIRC:PACKED ARRAY[1..?] OF CHAR;
                         F_DSIZ:INTEGER;
                     VAR F_TERM:PACKED ARRAY[1..?] OF CHAR;
                         F_TSIZ:INTEGER);
                     EXTERNAL;
              .
              .
              .
   DIRC::INTEGER:=-1; TERM::INTEGER:=-1;
   PX$CHF (FORM,DIRC,UB(DIRC),TERM,UB(TERM));
```

FORTRAN Example 1:

```
   C
   C A STANDARD CHANGE FORM CALL.
   C

                .
                .
                .
     CHARACTER CFORM*6,CDIRC*8,CTERM*4
     INTEGER DIRC(4),TERM(2)
     EQUIVALENCE (CDIRC,DIRC(1)),(CTERM,TERM(1))
     CFORM='FORM01'
     CDIRC='PATHNAME'
     CTERM='ST10'
                .
                .
                .
   C THE ASSIGNMENTS ABOVE ALSO DEFINE DIRC AND TERM
   C BECAUSE OF THE EQUIVALENCE
     CALL FFCHF (CFORM,DIRC(1),DIRC(4),TERM(1),TERM(2))
```

FORTRAN Example 2:

```
C
C  CHANGE ONLY THE FORM.  DIRECTORY, TERMINAL ARE UNCHANGED.
C
   CFORM='FORM02'
   CDIRC(1)=-1
   CTERM=-1
            .
            .
            .
   CALL FFCHF (CFORM,DIRC(1),DIRC(4),TERM(1),TERM(2))
```

5.7.5 Change Form Program Notes

Your application should check the status block after each Change Form command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.8 CHANGE ITC/IPC COMMUNICATION ROUTINE

This routine disconnects the application from the Executor it is using or it connects the application to an existing Executor. It enables one application task to pass an Executor/terminal to another application task without closing and reopening a form. Thus, the application can be written as a series of independent tasks to circumvent memory limitations.

A secondary use of this routine is to permit one application task to communicate serially with the Executors associated with two or more terminals.

The DX10 Form Executor uses the intertask communication facility (ITC). The DNOS Form Executor uses the interprocess communication facility (IPC). Though there are some differences between ITC and IPC, TIFORM treats them identically.

5.8.1 Change ITC/IPC Communication Calling Sequences

The calling sequences for the Change ITC/IPC Communication routine are as follows:

COBOL 3.1:

```
CALL "CF$CIC" USING < CIC argument> .
```

COBOL 3.2:

```
CALL "CX$CIC" USING < CIC argument> .
```

Pascal:

```
PX$CIC (VAR < CIC argument > :INTEGER);
```

FORTRAN:

```
CALL FFCIC (< CIC argument> )
```

5.8.2 Change ITC/IPC Communication Parameters

The < CIC argument> parameter is always an 01-level, signed, COMP-1 item in COBOL 3.1 or an INTEGER item in Pascal and FORTRAN. The function of the Change ITC/IPC Communication routine is determined by the value of < CIC argument> as follows:

- Value = – 1 breaks the connection between the application task and its Executor. The Executor is not terminated. The binary run ID of the Executor is returned in < CIC argument> . The application task must have a form open (that is, it must be connected to an Executor) to issue a – 1 Change ITC/IPC Communication routine.

- Value = positive integer between 0 and 255 establishes a connection between the application task and an existing, disconnected Executor. The value in < CIC argument> must be the binary run ID of the Executor to which the application is to be connected. The application must not be connected to an Executor when it issues a positive Change ITC/IPC Communication routine, and the Executor specified by the value of < CIC argument> must not be connected to any application task. The value of < CIC argument> is not changed by a positive Change ITC/IPC Communication routine.

5.8.3 Change ITC/IPC Communication Results

Status is posted. If < CIC argument> = −1 on the call, the task ID of the Executor with which communication is presently taking place is returned in < CIC argument> and communication with that Executor is suspended. If < CIC argument> is positive, communication with the Executor denoted by the run ID found in < CIC argument> is resumed.

5.8.4 Change ITC/IPC Communication Examples

The following examples demonstrate the Change ITC/IPC Communication routine:

COBOL 3.1: .

This example illustrates the serial processing of more than one terminal, using the COBOL 3.1 calls.

```
      .
      .
      .
 01 FORM           PIC X(6) VALUE "FORM01".
 01 DIR            PIC X(10) VALUE ".FORM.PATH".
 01 DIR-END        PIC XX.
*
 01 TERM1          PIC X(4) VALUE "ST01".
 01 TERM1-END      PIC XX.
 01 TERM2          PIC X(4) VALUE "ST02".
 01 TERM2-END      PIC XX.
*
 01 SEGMENT        PIC X(6) VALUE "SEGT01".
*
 01 TERM1-TASK-ID PIC S9(5) USAGE COMP-1.
 01 TERM2-TASK-ID PIC S9(5) USAGE COMP-1.
*
 01 GROUP          PIC X(6) VALUE "GROUP1".
 01 GRP-DATA       PIC X(18).
 01 GRP-DATA-END   PIC XX.
       .
       .
       .
*  START UP TERMINAL 1.
**********************
*
     CALL "CF$OF" USING FORM,
                        DIR, DIR-END,
                        TERM1, TERM1-END.
     CALL "CF$PS" USING SEGMENT.
*
* SUSPEND COMMUNICATION WITH TERMINAL 1.
****************************************
*
     MOVE -1 TO TERM1-TASK-ID.
     CALL "CF$CIC" USING TERM1-TASK-ID.
*
*  START UP TERMINAL 2.
**********************
*
```

```
     CALL "CF$OF" USING FORM,
                        DIR, DIR-END,
                        TERM2, TERM2-END.
     CALL "CF$PS" USING SEGMENT.
*
*  SUSPEND COMMUNICATION WITH TERMINAL 2.
****************************************
*
     MOVE -1 TO TERM2-TASK-ID.
     CALL "CF$CIC" USING TERM2-TASK-ID.
*
*  RE-ESTABLISH COMMUNICATION WITH TERMINAL 1.
*********************************************
*
     CALL "CF$CIC" USING TERM1-TASK-ID.
*
*  CONTINUE FORMS PROCESSING WITH TERMINAL 1.
********************************************
*
     CALL "CF$REA" USING GROUP, G-DATA, G-DATA-END.
```

COBOL 3.2:

This example shows one way that an Executor can be passed from task to task. A synonym is used to pass the Executor's run ID from one task to the other. Note that this is a COBOL 3.2 example using subroutines provided only by COBOL 3.2.1.

```
             .
             .
             .
*
* THE FIRST COBOL TASK.
***********************
 01  EXEC-ID       PIC S9(5) USAGE IS COMP-1.
 01  DISP-EXEC-ID  PIC  9(2).
 01  ERROR-CODE    PIC  99.
 01  EXEC-SYN      PIC  X(8) VALUE "EXECTASK".
 01  GROUP         PIC X(6) VALUE "GROUP1".
 01  WRITE-DATA    PIC X(23).
             .
             .
             .
     CALL CX$WRI USING GROUP, WRITE-DATA.
*
* DISCONNECT FROM AND GET THE TASK ID OF THE EXECUTOR.
******************************************************
*
     MOVE -1 TO EXEC-ID.
     CALL "CX$CIC" USING EXEC-ID.
*
*  SET A SYNONYM TO THE EXECUTOR'S TASK ID.
*******************************************
*
```

```
     MOVE EXEC-ID TO DISP-EXEC-ID.
     CALL "C$SETS"  USING ERROR-CODE, EXEC-SYN, DISP-EXEC-ID.
*
* BID THE SECOND COBOL TASK, THEN TERMINATE.
*******************************************
*
     CALL "C$CBID" USING ERR, SECOND-ID, LUNO, FLAGS.
     STOP RUN.
            .
            .
            .
**************************************************
**************************************************
*  THE SECOND COBOL TASK.
*************************
*
            .
            .
            .
 01  EXEC-SYN          PIC X(8) VALUE "EXECTASK".
 01  DISP-EXEC-ID      PIC 99.
 01  EXEC-ID           PIC S9(5) USAGE IS COMP-1.
 01  ERROR-CODE        PIC 99.
 01  STATUS-BLOCK.
     03  FORM-STATUS   PIC 9(2).
            .
            .
            .
 01 GROUP               PIC X(6) VALUE "GROUP1".
 01 READ-DATA           PIC X(23).
            .
            .
            .
* THE SECOND COBOL TASK GETS THE EXECUTOR'S SYNONYM
*     BY GETTING THE VALUE OF THE SYNONYM.
**************************************************
*
    CALL "C$MAPS" USING ERROR-CODE, EXEC-SYN, DISP-EXEC-ID.
    MOVE DISP-EXEC-ID TO EXEC-ID.
*
* RESUME PROCESSING WITH THE EXECUTOR.
**************************************
*
    CALL "CX$CIC" USING EXEC-ID.
*
* ESTABLISH THE STATUS BLOCK AND CONTINUE.
******************************************
*
    CALL "CX$STS" USING STATUS-BLOCK.
    CALL "CX$REA" USING GROUP, READ-DATA.
```

Pascal Example 1:

```
PROGRAM FIRST_TASK;

{The first Pascal task disconnects from the Executor and
gets its run ID, then sets a synonym to the ID value.}

            .
            .
            .
VAR EXEC_ID:INTEGER;
            .
            .
            .
PX$WRI (GROUP, WRITE_DATA, UB(WRITE_DATA));

{Disconnect from and get the task ID of the Executor.}

EXEC_ID:= -1;
PX$CIC (EXEC_ID);

{Set a synonym to the Executor's task ID with a
user-provided external procedure.}

SET_SYN (ERROR_CODE,EXEC_SYN,EXEC_ID);

{Bid the second Pascal task with a user-provided procedure,
then terminate.}

BID_TASK (ERR,SECOND_ID,LUNO,FLAGS);
ESCAPE FIRST_TASK;
            .
            .
            .
{================================================================}

{================================================================}

PROGRAM SECOND_TASK;


{The second Pascal task gets the Executor's run ID
by getting the value of the synonym with a user-
provided procedure.}

VAR EXEC_ID:INTEGER;
             .
             .

{Get the value of the synonym containing the Executor's
run ID.}

GET_SYN (ERROR_CODE,EXEC_SYN,EXEC_ID);

{Connect and resume processing with the Executor.}
```

```
PX$CIC (EXEC_ID);

{Establish the status block and continue.}

PX$STS (STATUS_BLOCK);

PX$REA (GROUP, READ_DATA, UB(READ_DATA);
            .
            .
            .
{================================================================
================================================================
================================================================}
```

Pascal Example 2:

This example illustrates the serial processing of more than one terminal with the same task.

```
            .
            .
            .
VAR TERM1_RUN_ID, TERM2_RUN_ID: INTEGER;
    DIRY:  PACKED ARRAY[1..10] OF CHAR;
    TERM1: PACKED ARRAY[1..4] OF CHAR;
    TERM2: PACKED ARRAY[1..4] OF CHAR;
            .
            .
            .
PROCEDURE PX$OF (FRM:S$6;
              VAR DIR:PACKED ARRAY[1..?] OF CHAR:
                 DIRSIZ:INTEGER;
             VAR TRM:PACKED ARRAY[1..?] OF CHAR;
                 TRMSIZ:INTEGER); EXTERNAL;

PROCEDURE PX$PS (SEG:S$6); EXTERNAL;

PROCEDURE PX$CIC (VAR CIC_ARG:INTEGER); EXTERNAL;

            .
            .
            .
DIRY:='.FORM.PROG';
TERM1:='ST01';
  {Start up terminal 1.}

PX$OF  ('FORMAA',DIRY,UB(DIRY),TERM1,UB(TERM1));

PX$PS  ('SEGT01');

  {Suspend communication with terminal 1.}

TERM1_RUN_ID:=-1;
PX$CIC (TERM1_RUN_ID);
```

```
  {Start up terminal 2.}
TERM2='ST02';
PX$OF  ('FORMAA,DIRY,UB(DIRY),TERM2,UB(TERM2));

PX$PS  ('SEGT01');

  {Suspend communication with terminal 2.}

TERM2_TASK_ID:=-1;
PX$CIC (TERM2_RUN_ID);

  {Reestablish communication with terminal 1.}

PX$CIC (TERM1_RUN_ID);

PX$REA (....
```

FORTRAN Example 1:

```
C
C       The first FORTRAN task disconnects from the Executor and
C       gets its run ID, then sets a synonym to the ID value.
C
                  .
                  .
                  .
        INTEGER EXECID
                  .
                  .
                  .
        CALL FFWRI (GNAME, WDATA(1), WDATA(END))
C
C       Disconnect from and get the task ID of the Executor.
C
        EXECID= -1
        CALL FFCIC (EXECID)
C
C       Set a synonym to the Executor's task ID with a
C       user-provided external procedure.
C
        CALL SETSYN (ERRCODE,EXECSYN,EXECID)
C
C       Bid the second FORTRAN task with a user-provided procedure,
C       then terminate.

        CALL BIDTSK(TASK,IERR)
                  .
                  .
                  .
C       ================================================================
C
C       ================================================================
C
```

```
C     The second FORTRAN task gets the Executor's run ID
C     by getting the value of the synonym with a user-
C     provided procedure.
C
      INTEGER EXECID
C                 .
C                 .
C                 .
C     Get the value of the synonym containing the Executor's
C     run ID.
C
      CALL GETSYN (ERRCODE,EXECSYN,EXECID)
C
C     Connect and resume processing with the Executor.
C
      CALL FFCIC (EXECID)
C
C     Establish the status block and continue.
C
      CALL FFSTS (STATUS)
C
      CALL FFREA (GNAME, RDATA(1), RDATA(END))
                 .
                 .
                 .
C
C     Continue executing form.
C
```

FORTRAN Example 2:

This example illustrates the serial processing of more than one terminal with the same task.

```
                 .
                 .
                 .
      INTEGER T1RUNID, T2RUNID
      CHARACTER*10 DIRY
      CHARACTER EDIRY
      CHARACTER*4 TERM1
      CHARACTER ETERM1
      CHARACTER*4 TERM2
      CHARACTER ETERM2
                 .
                 .
                 .
      DIRY='.FORM.PROG'
      TERM1='ST01'
C
C     Start up terminal 1.
C
      CALL FFOF ('FORMAA',DIRY,EDIRY,TERM1,ETERM1)
C
      CALL FFPS ('SEGT01')
```

```
C
C     Suspend communication with terminal 1.
C
      T1RUNID=-1
      CALL FFCIC (T1RUNID)
C
C     Start up terminal 2.
C
      TERM2='ST02'
      CALL FFOF ('FORMAA',DIRY,EDIRY,TERM2,ETERM2)
C
      CALL FFPS ('SEGT01')
C
C     Suspend communication with terminal 2.
C
      T2TASKID=-1
      CALL FFCIC (T2RUNID)
C
C     Reestablish communication with terminal 1.
C
      CALL FFCIC (T1RUNID)
C
      CALL FFREA (....
C
C     Continue processing the form.
C
```

5.8.5 Change ITC/IPC Communication Program Notes

Your application should check the status block after each Change ITC/IPC Communication command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.9 CLOSE FORM ROUTINE

This routine terminates form processing. To simplify error handling, a close can be done whether the form was successfully opened or not.

5.9.1 Close Form Calling Sequences

The calling sequences for the Close Form routine are as follows:

COBOL 3.1:

```
CALL "CF$CF".
```

COBOL 3.2:

```
CALL "CX$CF".
```

Pascal:

```
PX$CF;
```

FORTRAN:

```
CALL FFCF
```

5.9.2 Close Form Parameters

None.

5.9.3 Close Form Results

Status is posted. The form is closed. The terminal is released. All communication with the current instance of the Form Executor is ceased and the Executor is terminated.

5.9.4 Close Form Examples

The following examples demonstrate the Close Form routine:

COBOL 3.1:

```
      .
      .
      .
CALL "CF$CF".
```

Pascal:

```
          .
          .
          .
PROCEDURE PX$CF; EXTERNAL;
          .
          .
          .
PX$CF;
```

FORTRAN:

```
CALL FFCF
```

5.9.5 Close Form Program Notes

Your application should check the status block after each Close Form command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.10 CONTROL FUNCTIONS ROUTINE

This routine allows the application to specify various control functions to the Form Executor. The specified functions are performed immediately before control is returned to the application. Each control function is assigned an integer by which it is known. If the application issues a Control Functions routine with an argument of + N, the Executor is said to be in control mode N.

5.10.1 Control Functions Calling Sequences

The calling sequences for the Control Functions routine are as follows:

COBOL 3.1:

```
CALL "CF$CN" USING < control arg> .
```

COBOL 3.2:

```
CALL "CX$CN" USING < control arg> .
```

Pascal:

```
PX$CN (< control arg> );
```

FORTRAN:

```
CALL FFCN (< control arg> )
```

5.10.2 Control Functions Parameters

The < control arg> parameter is the 01-level, signed, COMP-1 item in COBOL 3.1, the signed integer item in Pascal, or the INTEGER*2 item in FORTRAN containing the control action code. If the code is + N, the Nth control action is turned on. If the code is – N, then the Nth control action is turned off. The legal control action codes and their meanings are described in the following paragraphs.

Codes 1 through 3 and 6 through 7 apply to all terminal device types, while codes 4 and 5 apply only to keyboard send/receive (KSR) devices and are ignored for other device types. See paragraph 2.9 for a complete description of forms processing of KSR device types.

Code Meanings

1 Disable the termination of a read on a Back Tab or Previous Field key when in the first field of the Read. When these keys are entered, sound a warning beep and position the cursor at the first character of the first field of the Read. This mode relieves the application of the responsibility of checking for form status 01 after each Read command.

2 Disable the Erase Input key; allow it to perform like Erase Field. Also, clear a message from the message area of the screen immediately upon acknowledgement by the user. This mode is used by the ISGE and is not particularly useful for a general application.

3 Disable deblanking from the right for field and variable values.

4 Place the KSR terminal in the immediate write mode. Usually, the terminal is in the delayed write mode.

5 I/O on the KSR terminal is to be unformatted. Normally, I/O is to be formatted.

6 If no data is entered, return ASCII nulls instead of blanks. The default is for the Executor to return blanks.

7 When enabled, the Previous Line and Next Line keys are forced to move the cursor to the first column of the selected field. If off, the Previous Line and Next Line keys leave the cursor in the same screen column that it started in. This mode is especially useful when dealing with columnar numeric data.

8 When enabled, performs symmetric processing to the application.

9 When enabled, suppresses the Print key message to the screen.

10* When enabled, allows an open extend on the 820. User's 820 files are concatenated.

11 When enabled, validation execution takes place on a Previous Field key out of the first field in a read.

12 When enabled and Print Key task is being used, inhibits the printing of header and trailer lines for screens.

13* When enabled, allows an open extend on the 820.

5.10.3 Control Functions Results

Status is posted.

5.10.4 Control Functions Examples

The following examples demonstrate the Control Functions routine:

COBOL 3.1:

```
          .
          .
01 CONT-ARG     PIC S9(5) COMP-1.
          .
          .
   MOVE 1 TO CONT-ARG.
          .
          .
   CALL "CF$CN" USING CONT-ARG.
          .
          .
   MOVE -1 TO CONT-ARG.
          .
          .
   CALL "CF$CN" USING CONT-ARG.
```

*See Program Notes.

Pascal:

```
            .
            .
VAR CONDITION:INTEGER;
            .
            .
PROCEDURE PX$CN (VAR COND:INTEGER); EXTERNAL;
            .
            .
CONDITION:=1;
PX$CN (CONDITION);
            .
            .
CONDITION:=-1;
PX$CN (CONDITION);
```

FORTRAN:

```
            .
            .
INTEGER CONDITION
            .
            .
CONDITION=1
CALL FFCN (CONDITION)
            .
            .
CONDITION=-1
CALL FFCN (CONDITION)
```

5.10.5 Control Functions Program Notes

Your application should check the status block after each Control Functions command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

Both control modes 10 and 13 must be activated as follows:

1. Open the form.
2. Issue the control mode.
3. Change the form. Explicitly declare each parameter the same as in the Open Form. Do not use defaults.

5.11 DECLARE STATUS BLOCK ROUTINE

This routine must be the first routine issued and may be reissued at any time thereafter. Its purpose is to notify the interface routines of the address in memory assigned by the application for construction of the 40-byte status block. This block must begin on a word boundary. Therefore, if you are using COBOL 3.1, you must declare this block at the 01 level. The interface routines save the status block address specified on the last Declare Status Block call. The status of each subsequent call is placed in the block at this address, formatted as shown and discussed in paragraph 5.4.

5.11.1 Declare Status Block Calling Sequences

The calling sequences for the Declare Status Block routine are as follows:

COBOL 3.1:

CALL "CF$STS" USING < status block> .

COBOL 3.2:

CALL "CX$STS" USING < status block> .

Pascal:

PX$STS (VAR < status block> :STATUS_BLOCK);

FORTRAN:

CALL FFSTS (< status block>)

5.11.2 Declare Status Block Parameters

The < status block> parameter is the 40-byte long item (01-level item in COBOL 3.1) where the interface routines construct the status block.

5.11.3 Declare Status Block Results

When your program issues the Declare Status Block routine, TIFORM can access the status block and return the appropriate status values after subsequent routine calls.

5.11.4 Declare Status Block Examples

The following examples demonstrate the Declare Status Block routine:

COBOL 3.1

```
*
*    ...PICK UP STATUS BLOCK DEF'N FROM COPY LIB...
*
      COPY CLIB.C$STATUS.
*
*    FIELDS WITHIN TIFORM'S STATUS BLOCK...
*
*01  TIFORM-STATUS-BLOCK.
*      05  FORM-STATUS              PIC 9(02).
*      05  OPSYS-STATUS             PIC X(02).
```

```
*       05  EVENT-KEY              PIC 9(02).
*       05  COMMAND                PIC X(02).
*       05  ITEM-NAME              PIC X(06).
*       05  INDEX-01               PIC 9(03).
*       05  INDEX-02               PIC 9(03).
*       05  ITEM-CNT               PIC 9(03).
*       05  CURSOR-POSITION        PIC 9(03).
*       05  TEXT-LENGTH            PIC 9(05)
*                                   USAGE IS COMP-1.
*       05  FILLER                 PIC X(12).
                    .
                    .
                    .
    CALL "CF$STS" USING TIFORM-STATUS-BLOCK.
                    .
                    .
                    .
```

Pascal:

```
                .
                .
                .
    TYPE
        C$2 = PACKED ARRAY[1..2] OF CHAR;
        C$6 = PACKED ARRAY[1..6] OF CHAR;
                .
                .
                .
        STATUS_BLOCK = PACKED RECORD
            FORM_STATUS:C$2;
            OPSYS_STATUS:C$2;
            EVENT_KEY:C$2;
            COMMAND:C$2;
            ITEM_NAME:C$6;
            INDEXES_CURSOR:PACKED ARRAY[1..12] OF CHAR;
            TEXT_LENGTH: INTEGER;
            FILLER_1:PACKED ARRAY [1..12] OF CHAR;
            END;
                .
                .
                .
    VAR SBLOCK:STATUS_BLOCK;
                .
                .
                .
    PROCEDURE PX$STS(VAR SBLK:STATUS_BLOCK); EXTERNAL;
                .
                .
                .
    PX$STS (SBLOCK);{Declare status block.}
```

FORTRAN:

```
      INTEGER SBLOCK(20),TLENGTH
      CHARACTER*2 STATUS,OPSYS,EVENT,COMAND
      CHARACTER*12 INDCUR,FILL
      CHARACTER*6 ITEM
*
      EQUIVALENCE (SBLOCK(1),STATUS),
     C            (SBLOCK(2),OPSYS),
     C            (SBLOCK(3),EVENT),
     C            (SBLOCK(4),COMAND),
     C            (SBLOCK(5),ITEM),
     C            (SBLOCK(8),INDCUR),
     C            (SBLOCK(14),TLENGTH),
     C            (SBLOCK(15),FILL)
```

5.11.5 Declare Status Block Program Notes

If you fail to declare the status block, errors occur at run-time.

5.12 DISARM EVENT KEYS ROUTINE

This routine disarms all function keys. It is precisely equivalent to an Arm Event Keys call with no keys specified.

5.12.1 Disarm Event Keys Calling Sequences

The calling sequences for the Disarm Event Keys routine are as follows:

COBOL 3.1:

```
CALL "CF$DAK".
```

COBOL 3.2:

```
CALL "CX$DAK".
```

Pascal:

```
PX$DAK;
```

FORTRAN:

```
CALL FFDAK
```

5.12.2 Disarm Event Keys Parameters

None.

5.12.3 Disarm Event Keys Results

Status is posted. All function keys are disarmed.

5.12.4 Disarm Event Keys Examples

The following examples demonstrate the Disarm Event Keys routine:

COBOL 3.1:

```
        .
        .
        .
CALL "CF$DAK".
```

Pascal:

```
            .
            .
            .
PROCEDURE PX$DAK;  EXTERNAL;
            .
            .
            .
PX$DAK;
```

FORTRAN:

```
CALL FFDAK
```

5.12.5 Disarm Event Keys Program Notes

Your application should check the status block after each Disarm Event Keys command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.13 EXECUTE ASYNCHRONOUSLY ROUTINE

The Execute Asynchronously routine forces the next routine issued to execute asynchronously. That is, for the routine that follows the Execute Asynchronously routine, control is returned to the application immediately after the interface routine has sent it to the Executor. The interface routine does not wait for a reply from the Executor. Thus, the Execute Asynchronously routine permits the application to continue its processing while the Executor processes the next routine.

The Execute Asynchronously routine affects only a single routine, the routine issued immediately following. Subsequent routines are executed synchronously.

The status of an asynchronously executed routine is lost unless it is followed by the Execute Asynchronously routine.

5.13.1 Execute Asynchronously Calling Sequences

The calling sequences for the Execute Asynchronously routine are as follows:

COBOL 3.1:

```
CALL "CF$ASN".
```

COBOL 3.2:

```
CALL "CX$ASN".
```

Pascal:

```
PX$ASN;
```

FORTRAN:

```
CALL FFASN
```

5.13.2 Execute Asynchronously Parameters

None.

5.13.3 Execute Asynchronously Results

Following the issuance of the next TIFORM routine, control is returned immediately to the application without waiting for the completion of processing of that routine by the Executor.

5.13.4 Execute Asynchronously Examples

The following examples demonstrate the Execute Asynchronously routine:

COBOL 3.1:

```
                .
                .
                .
*
* THE READ CALL IS TO BE EXECUTED ASYNCHRONOUSLY.
************************************************
*
     CALL "CF$ASN".
     CALL "CF$REA" USING GRP, R-DATA, R-DATA-END.
*
* PERFORM UNRELATED PROCESSING.
*******************************
*
     PERFORM DO-SOMETHING-ELSE.
*
* SYNCHRONIZE APPLICATION AND EXECUTOR.
***************************************
*
     CALL "CF$SYN".
*
* PROCESS THE RESULTS OF THE READ.
*
  PERFORM STATUS-CHECK.
```

Pascal:

```
                .
                .
                .
PROCEDURE PX$ASN; EXTERNAL;
PROCEDURE PX$SYN; EXTERNAL;
PROCEDURE DO_SOMETHING_ELSE;
                .
                .
                .
{ THE READ CALL IS TO BE EXECUTED ASYNCHRONOUSLY.
 ================================================}
  PX$ASN;
  PX$REA (GROUP,R_DATA,UB(R_DATA));

{ PERFORM UNRELATED PROCESSING.
 ==============================}

   DO_SOMETHING_ELSE;

{ SYNCHRONIZE APPLICATION AND EXECUTOR.
 ======================================}

   PX$SYN;

{PERFORM STATUS CHECK ON PREVIOUS COMMAND.}

{ PROCESS THE RESULTS OF THE READ.}
```

FORTRAN:

```
C
C   THE READ CALL IS TO BE EXECUTED ASYNCHRONOUSLY.
C
      CALL FFASN
      CALL FFREA (GNAME,RDATA(1),RDATA(END))
C
C   PERFORM UNRELATED PROCESSING.
C
      CALL UNRELATE
C
C   SYNCHRONIZE APPLICATION AND EXECUTOR.
C

      CALL FFSYN
C
C  PERFORM STATUS CHECK ON PREVIOUS COMMAND.
C
C   PROCESS THE RESULTS OF THE READ.
```

5.13.5 Execute Asynchronously Program Notes

Your application should check the status block after each Execute Asynchronously command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.14 OPEN FORM ROUTINE

This routine accepts a form name, a program file name, and a terminal name. It validates the names, bids the Form Executor, and passes the names to it. An Open Form must be issued before any other TIFORM function, other than Declare Status Block, can be performed.

The form directory name and the terminal name are both specified to this routine by their starting and ending addresses. If the actual text of these names is shorter than the space specified, they must be blank-filled on the right. Synonyms may be used in these two parameters. The synonym resolution is done by the interface routines prior to passing the names to the Executor, using the application"s synonyms. All the rules of synonym resolution in pathnames apply.

Upon return, status is posted in the status block, the form is marked OPEN (if there have been no errors), and control is returned to the application.

5.14.1 Open Form Calling Sequences

The calling sequences for the Open Form routine are as follows:

COBOL 3.1:

```
CALL "CF$OF" USING <form name>, <directory name>, <directory name end>,
                   <terminal name>, <terminal name end>.
```

COBOL 3.2:

```
CALL "CX$OF" USING <form name>, <directory name>, <terminal name>.
```

Pascal:

```
PX$OF (<form name>, <directory name>, <directory name size>, <terminal name>,
       <terminal name size>);
```

FORTRAN:

```
      CALL FFOF (<form name>, <directory name>, <directory name end>,
     1           <terminal name>, <terminal name end>)
```

5.14.2 Open Form Parameters

The following list describes the parameters for the Open Form routine:

<form name> is the six-byte item containing the name of the form to be opened. (This item is a 01-level item in COBOL 3.1, a CHAR item in Pascal, and a CHARACTER*6 item in FORTRAN.) The name of the form must be left-justified, blank-filled on the right within this item.

<directory name> represents the starting address of the 2 – 48 byte directory name area in which the pathname/synonym specifying the program file in which the specified form is stored must reside. The entry in this field must be left-justified and blank-filled on the right. If the first byte of the area is a blank, the synonym DIRECTRY is assumed to point to the program file.

< directory name end> in COBOL 3.1 and FORTRAN represents the end address of the pathname area.

< directory name size> in Pascal represents the size of the pathname area.

< terminal name> represents the start address of the 2 – 48 byte terminal/file name area in which the device name or file pathname (or a synonym specifying the terminal or file) on which the Form Executor is to execute the specified form must reside. The entry in this field must be left-justified and blank-filled on the right. If the first byte of the area is a blank, then the synonym ME (pointing to the application's controlling station) is used, causing the form to be executed on the application's own station.

< terminal name end> in COBOL 3.1 and FORTRAN represents the end address of the terminal name area.

< terminal name size> in Pascal represents the size of the terminal name area.

5.14.3 Open Form Results

Status is posted. The Form Executor is put into execution for this application. The Form Executor acquires the specified terminal and opens the specified form.

5.14.4 Open Form Examples

The following examples demonstrate the Open Form routine:

COBOL 3.1:

```
          .
          .
          .
 01   FORM-NAME PIC X(6) VALUE IS "FORM1".
*
* FORCE THE USE OF THE "DIRECTRY" SYNONYM...
*
 01   DIRECTORY-NAME     PIC X(2) VALUE IS SPACES.
 01   DIRECTORY-NAME-END PIC X(2).
*
* EXPLICITLY SPECIFY STATION #5...
*
 01   TERMINAL-NAME      PIC X(8) VALUE IS "ST05".
 01   TERMINAL-NAME-END  PIC X(2).
          .
          .
          .
      CALL "CF$OF" USING FORMNAME,
                         DIRECTORY-NAME,
                         DIRECTORY-NAME-END,
                         TERMINAL-NAME,
                         TERMINAL-NAME-END.
```

Pascal:

```
                .
                .
                .
   DIR_NAME:PACKED ARRAY[1..10] OF CHAR;

   TRM_NAME:PACKED ARRAY[1..4] OF CHAR;

   VAR D_NAME:DIR_NAME;
       T_NAME:TRM_NAME;
                .
                .
                .
   PROCEDURE PX$OF (FNAME:C$6;
                    VAR DIRNME:PACKED ARRAY[1..?] OF CHAR;
                        DIRSIZ:INTEGER;
                    VAR TRMNME:PACKED ARRAY[1..?] OF CHAR;
                        TRMSIZ:INTEGER);
                    EXTERNAL;
                .
                .
                .
   D_NAME:='.FORM.FILE'; {Full pathname.}
   T_NAME:='TERM';       {Synonym.}
                         {Note literal is used for form name.}
   PX$OF ('FORM01', D_NAME,UB(D_NAME),T_NAME,UB(T_NAME));
```

FORTRAN:

```
     CHARACTER*2 DRCTRY(2), TRMNL(2),
          .
          .
          .
     CALL FFOF ('FORM01', DRCTRY(1), DRCTRY(2),
    1             TRMNL(1), TRMNL(2))
```

5.14.5 Open Form Program Notes

Your application should examine the status block after every Open command and take the appropriate end action in case of errors. You can only open one form at a time. To open a second form, you must close the first form or use the Change Form routine, which is a combination Close and Open Form.

5.15 PREPARE SEGMENT ROUTINE

This routine causes a specified segment of the currently open form to be loaded into the Form Executor, its background mask to be displayed, and all default values of the segment's fields to be displayed. This routine must be issued before any Read, Write, or Reset routine affecting the groups, fields, or variables of a segment can be issued.

This routine causes the Form Executor to discard whatever segment is active when the routine is issued. It also causes all event key arming to be reset.

5.15.1 Prepare Segment Calling Sequences

The calling sequences for the Prepare Segment routine are as follows:

COBOL 3.1:

CALL "CF$PS" USING < segment name> .

COBOL 3.2:

CALL "CX$PS" USING < segment name> .

Pascal:

PX$PS (< segment name>);

FORTRAN:

CALL FFPS (< segment name>)

5.15.2 Prepare Seqment Parameters

The < segment name> parameter is a six-byte item (01-level item in COBOL 3.1) containing the name of the segment of the current form to make active. The name must be left-justified and blank-filled on the right within this item.

5.15.3 Prepare Segment Results

Status is posted. The specified segment is loaded into the Form Executor and made active. The background and default values of the segment's fields are displayed.

5.15.4 Prepare Segment Examples

The following examples demonstrate the Prepare Segment routine:

COBOL 3.1:

```
          .
          .
          .
01   SEG-NAME PIC X(6) VALUE IS "SGNAME".
          .
          .
          .
CALL "CF$PS" USING SEG-NAME.
```

Pascal:

```
                .
                .
                .
   VAR SEGMENT:C$6;
                .
                .
                .
   PROCEDURE PX$PS (SNAME:C$6); EXTERNAL;
                .
                .
                .
   SEGMENT:='SEG001';
   PX$PS (SEGMENT);
```

FORTRAN:

```
                .
                .
                .
   CHARACTER SNAME*6
                .
                .
                .
   SNAME='SEG001'
   CALL FFPS (SNAME)
```

5.15.5 Prepare Segment Program Notes

Your application should check the status block after each Prepare Segment command. If an error occurs, your application should report the error to the user in order to identify the problem. Note that there is no close segment operation. The Prepare Segment command automatically closes the current segment (if any) before preparing the new segment.

5.16 PRINT KEY ROUTINE

This routine prints the current contents of the screen (or the virtual screen for a KSR device) on the terminal's associated printer or queues the contents for later printing if that printer is busy. This routine provides the application program with the same printing capability that is available to the terminal user via the Print key. Paragraph 2.11 describes the operation of the Print key.

5.16.1 Print Key Calling Sequences

The calling sequences for the Print Key routine are as follows:

COBOL 3.1:

```
CALL "CF$PKY".
```

COBOL 3.2

```
CALL "CX$PKY".
```

Pascal:

```
PX$PKY;
```

FORTRAN:

```
CALL FFPKY
```

5.16.2 Print Key Parameters

None.

5.16.3 Print Key Results

The current VDT screen image or KSR virtual screen image is written to a file that is either immediately printed or, if the printer is busy, is queued for later printing.

5.16.4 Print Key Examples

The following examples demonstrate the Print Key routine:

COBOL 3.1:

```
        .
        .
        .
CALL "CF$PKY".
```

Pascal:

```
                .
                .
                .
PROCEDURE PX$PKY; EXTERNAL;
                .
                .
                .
PX$PKY;
```

FORTRAN:

```
CALL FFPKY
```

5.16.5 Print Key Program Notes

Your application should check the status block after each Print Key command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.17 READ A GROUP ROUTINE

This routine performs a read operation on the entire group named. See paragraph 1.4.3 for a discussion of the functions of a group read.

5.17.1 Read a Group Calling Sequences

The calling sequences for the Read a Group routine are as follows:

COBOL 3.1:

CALL "CF$REA" USING < group name> , < read data> , < read data end> .

COBOL 3.2:

CALL "CX$REA" USING < group name> , < read data> .

Pascal:

PX$REA (< group name> ,< read data> , < read data size>);

FORTRAN:

CALL FFREA (< group name> , < read data> , < read data end>)

5.17.2 Read a Group Parameters

The following list describes the parameters for the Read a Group routine:

< group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group, field, or variable to read. The name must be left-justified and blank-filled on the right within this item.

< read data> is an item (01-level item in COBOL 3.1) into which the group's elemental members values are read.

< read data end> is an item (01-level item in COBOL 3.1) indicating the end of the read data area.

< read data size> is an item indicating the size of the read data area.

5.17.3 Read a Group Results

Status is posted. The group's elemental members are read and their values are placed in the read data area.

5.17.4 Read a Group Examples

The following examples demonstrate the Read a Group routine:

COBOL 3.1:

```
          .
          .
          .
01  GROUP-NAME PIC X(6) VALUE IS 'NAME02'.
01  READ-DATA      PIC X(80).
01  READ-DATA-END  PIC X(2).
          .
          .
          .
    CALL "CF$REA" USING GROUP-NAME,
                        READ-DATA, READ-DATA-END.
```

Pascal:

```
              .
              .
              .
VAR GROUP:C$6
    R_DATA:PACKED ARRAY[1..30] OF CHAR;
              .
              .
              .
PROCEDURE PX$REA (GRPNAM:C$6;
                  VAR READAT:PACKED ARRAY[1..?] OF CHAR;
                      REASIZ:INTEGER);  EXTERNAL;
              .
              .
              .
GROUP:='GROUP1';
PX$REA (GROUP,R_DATA,UB(R_DATA));
```

FORTRAN:

```
 CHARACTER GNAME*6,RDATA*2(40)
              .
              .
              .
C Read group one.
 GNAME='GROUP1'
              .
              .
              .
 CALL FFREA (GNAME,RDATA(1),RDATA(40))
```

5.17.5 Read a Group Program Notes

Your application should check the status block after each Read a Group command. After a read operation, the status block returns several nonzero codes that are not errors. For instance, 01 indicates a partial read terminated by exiting the first field backward. Depending on the requirements of your application, you may need to treat non-fatal (less than 10) status codes individually.

5.18 READ INDEXED ROUTINE

This routine reads a portion of a group as specified by the indexing parameters. These parameters are not altered by the routine. For information regarding group indexing, see paragraph 1.4.3.

5.18.1 Read Indexed Calling Sequences

The calling sequences for the Read Indexed routine are as follows:

COBOL 3.1:

```
CALL "CF$REX" USING <group name>, <read data>, <read data end>, <index-1>,
                    <index-2>, <count>, <cursor position>.
```

COBOL 3.2:

```
CALL "CX$REX" USING <group name>, <read data>, <index-1>, <index-2>, <count>,
                    <cursor position>.
```

Pascal:

```
PX$REX (<group name>, <read data>, <read data size>, <index-1>, <index-2>,
        <count>, <cursor position>);
```

FORTRAN:

```
      CALL FFREX (<group name>, <read data>, <read data end>, <index-1>, <index-2>,
     1            <count>, <cursor position>)
```

5.18.2 Read Indexed Parameters

The following list describes the parameters for the Read Indexed routine:

<group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to read.

<read data> is an item (01-level item in COBOL 3.1) into which the group's elemental members' values are placed.

<read data end> is an item (01-level item in COBOL 3.1) indicating the end of the read buffer.

<read data size> in Pascal is an item indicating the size of the read buffer.

<index-1> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the first-level index of the group's first elemental member to read.

<index-2> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the second-level index of the group's first elemental member to be read.

< count> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the number of elemental members to read.

< cursor position> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the initial cursor position in the indexed field.

5.18.3 Read Indexed Results

Status is posted. The indexed elements are read and their values are placed in the read data area.

5.18.4 Read Indexed Examples

The following examples demonstrate the Read Indexed routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME PIC X(6) VALUE IS "NAMEXX".
01   DATA-BEGIN PIC X(80).
01   DATA-END   PIC XX.
01   INDEX-1    PIC 999.
01   INDEX-2    PIC 999.
01   COUNT      PIC 999.
01   CURSOR-POS PIC 999.
          .
          .
          .
     MOVE 03 TO INDEX-1.
     MOVE 09 TO INDEX-2.
     MOVE 06 TO COUNT, CURSOR-POS.
     CALL "CF$REX" USING GROUP-NAME,
                         DATA-BEGIN, DATA-END,
                         INDEX-1, INDEX-2, COUNT,
                         CURSOR-POS.
```

Pascal:

```
            .
            .
            .
VAR GROUP:C$6;
    R_DATA:PACKED ARRAY[1..14] OF CHAR;
    X_1,X_2,COUNT,CR_POS:C$3;
            .
            .
            .
PROCEDURE PX$REX (GRP:C$6
                  VAR READAT:PACKED ARRAY[1..?] OF CHAR;
                      REASIZ:INTEGER;
                  VAR X1,X2,CNT,CPOS:C$3); EXTERNAL;
            .
            .
            .
{Read the 2nd through 4th elements of the first row of the array.}
GROUP:='ARY1  ';
X_1:='001'; X_2:='002'; COUNT:='003'; CR_POS:='000';
PX$REX(GROUP,R_DATA,UB(R_DATA),X_1,X_2,COUNT,CR_POS);
```

FORTRAN:

```
      CHARACTER GNAME*6,RDATA*80,REND*2
      CHARACTER*3 INDEX1,INDEX2,COUNT,CURPOS
                .
                .
                .
C     Read the 2nd through 4th elements of the first row of the array.
      GROUP='ARY1  '
      INDEX1='001'
      INDEX2='002'
      COUNT='003'
      CURPOS='000'
                .
                .
                .
      CALL FFREX (GNAME,RDATA(1),RDATA(40),INDEX1,INDEX2,COUNT,CURPOS)
```

5.18.5 Read Indexed Program Notes

Your application should check the status block after each Read Indexed command. After a read operation, the status block returns several nonzero codes that are not errors. For instance, 01 indicates a partial read terminated by exiting the first field backward. Depending on the requirements of your application, you may need to treat non-fatal (less than 10) status codes individually.

5.19 READ INDEXED WITH CURSOR RETURN ROUTINE

This routine is the same as Read Indexed except that the indexing parameters are altered to indicate which field the cursor was in when screen input was terminated. The indexing parameters point to the last field read. If the Read was terminated by an armed event key or the Enter key, the indexing parameters indicate the final position of the cursor.

5.19.1 Read Indexed With Cursor Return Calling Sequences

The calling sequences for the Read Indexed With Cursor Return routine are as follows:

COBOL 3.1:

```
CALL "CF$RXC" USING <group name>, <read data>, <read data end>, <index-1>,
                    <index-2>, <count>, <cursor position>.
```

COBOL 3.2:

```
CALL "CX$RXC" USING <group name>, <read data>, <index-1>, <index-2>, <count>,
                    <cursor position>.
```

Pascal:

```
PX$RXC (<group name>, <read data>, <read data size>, <index-1>, <index-2>,
        <count>, <cursor position>);
```

FORTRAN:

```
      CALL FFRXC (<group name>, <read data>, <read data end>, <index-1>, <index-2>,
     1            <count>, <cursor position>)
```

5.19.2 Read Indexed With Cursor Return Parameters

The following list describes the parameters for the Read Indexed With Cursor Return routine:

<group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to read.

<read data> is an item (01-level item in COBOL 3.1) into which the group's elemental members' values are placed.

<read data end> is an item (01-level item in COBOL 3.1) indicating the end of the read buffer.

<read data size> in Pascal is an item indicating the size of the read buffer.

<index-1> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the first level index of the group's first elemental member to read. The first-level index of the field containing the cursor is returned in this parameter.

<index-2> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the second-level index of the group's first elemental member to read. The second-level index of the field containing the cursor is returned in this parameter.

< count> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the number of elemental members to read. The elemental member number of the field containing the cursor is returned in this parameter.

< cursor position> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the character to start reading within the indexed field. The position of the cursor within the current field is returned in this parameter.

5.19.3 Read Indexed With Cursor Return Results

Status is posted. The elemental members' values are read into the read area. The cursor position is returned in the indexing parameters.

5.19.4 Read Indexed With Cursor Return Examples

The following examples demonstrate the Read Indexed With Cursor Return routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME PIC X(6) VALUE IS "NAMEXX".
01   DATA-BEGIN PIC X(80).
01   DATA-END   PIC X(2).
01   INDEX-1    PIC 999.
01   INDEX-2    PIC 999.
01   COUNT      PIC 999.
01   CURSOR-POS PIC 999.
          .
          .
          .
     MOVE 03 TO INDEX-1.
     MOVE 09 TO INDEX-2.
     MOVE 06 TO COUNT.
     CALL "CF$RXC" USING GROUP-NAME,
                         DATA-BEGIN, DATA-END,
                         INDEX-1, INDEX-2, COUNT,
                         CURSOR-POS.
```

Pascal:

```
            .
            .
            .
VAR GROUP:C$6;
    R_DATA:PACKED ARRAY[1..14] OF CHAR;
    X_1,X_2,COUNT,CR_POS:C$3;
            .
            .
            .
PROCEDURE PX$RXC (GRP:C$6
                  VAR READAT:PACKED ARRAY[1..?] OF CHAR;
                      REASIZ:INTEGER;
                  VAR X1,X2,CNT,CPOS:C$3); EXTERNAL;
            .
            .
            .
{Read the second through fourth elements of the first row of
    the array, and get the cursor position.}
GROUP:='ARAY  ';
X_1:='001'; X_2:='002'; CNT:='003';
PX$RXC(GROUP,R_DATA,UB(R_DATA),X_1,X_2,COUNT,CR_POS);
```

FORTRAN:

```
      CHARACTER GNAME*6,RDATA*2(40)
      CHARACTER*3 INDEX1,INDEX2,COUNT,CURPOS

                .
                .
                .
C     Read the second through fourth elements of the first row of
          the array, and get the cursor position.

      GNAME='ARAY  '
      INDEX1='001'
      INDEX2='002'
      COUNT='003'
      CURPOS='000'
                .
                .
                .
      CALL FFRXC (GNAME,RDATA(1),RDATA(40),INDEX1,INDEX2,COUNT,CURPOS)
```

5.19.5 Read Indexed With Cursor Return Program Notes

Your application should check the status block after each Read Indexed With Cursor Return command. After a read operation, the status block returns several nonzero codes that are not errors. For instance, 01 indicates a partial read terminated by exiting the first field backward. Depending on the requirements of your application, you may need to treat non-fatal (less than 10) status codes individually.

5.20 RESET FORM INDEXED ROUTINE

This routine is the same as Reset Form except that the index and count parameters can be used to specify a portion of a group. For information regarding group indexing, see paragraph 1.4.3.

5.20.1 Reset Form Indexed Calling Sequences

The calling sequences for the Reset Form Indexed routine are as follows:

COBOL 3.1:

CALL "CF$RFX" USING <group name>, <index-1>, <index-2>, <count>.

COBOL 3.2:

CALL "CX$RFX" USING <group name>, <index-1>, <index-2>, <count>.

Pascal:

PX$RFX (<group name>, <index-1>, <index-2>, <count>);

FORTRAN:

CALL FFRFX (<group name>, <index-1>, <index-2>, <count>)

5.20.2 Reset Form Indexed Parameters

The following list describes the parameters for the Reset Form Indexed routine:

<group name> is the six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to reset to its default value.

<index-1> is the three-byte item (01-level numeric item in COBOL 3.1) containing the first index of the first member of the named group to reset.

<index-2> is the three-byte item (01-level numeric item in COBOL 3.1) containing the second index of the first member of the named group to reset.

<count> is the three-byte item (01-level numeric item in COBOL 3.1) containing the number of elemental members to reset.

5.20.3 Reset Form, Indexed Results

Status is posted. The specified elemental members of the specified group are reset to their default values.

5.20.4 Reset Form Indexed Examples

The following examples demonstrate the Reset Form Indexed routine:

COBOL 3.1:

```
          .
          .
          .
01  GROUP-NAME  PIC X(6) VALUE IS "NAME99".
01  INDEX-1     PIC 999.
01  INDEX-2     PIC 999.
01  COUNT       PIC 999.
          .
          .
          .
    MOVE 2 TO INDEX-1.
    MOVE 2 TO INDEX-2.
    CALL "CF$RFX" USING GROUP-NAME, INDEX-1, INDEX-2.
```

Pascal:

```
              .
              .
              .
VAR GROUP:C$6;
    X_1,X_2,COUNT:C$3;
              .
              .
              .
PROCEDURE PX$RFX (GRP:C$6; VAR X1,X2,CN:C$3); EXTERNAL;
              .
              .
              .
{Reset four fields beginning with row two, column three.}
GROUP:='ARAY  ';
X_1:='002'; X_2:='003'; COUNT:='004';
PX$RFX (GROUP,X_1,X_2,COUNT);
```

FORTRAN:

```
                .
                .
                .
  CHARACTER GNAME*6
  CHARACTER*3 INDEX1,INDEX2,COUNT
                .
                .
                .
C Reset four fields beginning with row two, column three.
  GROUP='ARAY  '
  INDEX1='002'
  INDEX2='003'
  COUNT='004'
  CALL FFRFX (GROUP,INDEX1,INDEX2,COUNT)
```

5.20.5 Reset Form Indexed Program Notes

Your application should check the status block after each Disarm Event Keys command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.21 RESET FORM ROUTINE

This routine reinitializes all the fields in the specified group to their default values.

5.21.1 Reset Form Calling Sequences

The calling sequences for the Reset Form routine are as follows:

COBOL 3.1:

CALL "CF$RF" USING < group name> .

COBOL 3.2:

CALL "CX$RF" USING < group name> .

Pascal:

PX$RF (< group name>);

FORTRAN:

CALL FFRF (< group name>)

5.21.2 Reset Form Parameters

The < group name> parameter is the six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to reset to its default value.

5.21.3 Reset Form Results

Status is posted. All the elemental members of the specified group are reset to their default values.

5.21.4 Reset Form Examples

The following examples demonstrate the Reset Form routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME    PIC X(6) VALUE IS "NAMEYY"
          .
          .
          .
     CALL "CF$RF" USING GROUP-NAME.
```

Pascal:

```
              .
              .
              .
VAR GROUP:C$6;
              .
              .
              .
PROCEDURE PX$RF (GRP:C$6); EXTERNAL;
              .
              .
              .
GROUP:='GROUP8';
PX$RF (GROUP);
```

FORTRAN:

```
              .
              .
              .
CHARACTER*6 GNAME
              .
              .
              .
GROUP='GROUP8'
CALL PX$RF (GROUP)
```

5.21.5 Reset Form Program Notes

Your application should check the status block after each Reset Form command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.22 SYNCHRONIZE ROUTINE

This routine synchronizes the operation of the application and the Executor following the asynchronous execution of the previous routine. Because the Synchronize routine itself does not affect the status block, it allows the application to receive and examine the statuses of an asynchronously executed routine.

5.22.1 Synchronize Calling Sequences

The calling sequences for the Synchronize routine are as follows:

COBOL 3.1:

```
CALL "CF$SYN".
```

COBOL 3.2:

```
CALL "CX$SYN".
```

Pascal:

```
PX$SYN;
```

FORTRAN:

```
CALL FFSYN
```

5.22.2 Synchronize Parameters

None.

5.22.3 Synchronize Results

If the previous routine was executed asynchronously, the application is suspended until the Executor completes the processing of the previous routine. Results of the routine are returned to the application, and the application is restarted. The status that is posted following the execution of this routine is the status of the asynchronously executed routine. If the previous routine was not executed asynchronously, the Synchronize routine has no effect.

5.22.4 Synchronize Examples

The following examples demonstrate the Synchronize routine:

COBOL 3.1:

```
             .
             .
             .
*
* THE READ CALL IS TO BE EXECUTED ASYNCHRONOUSLY.
************************************************
*
     CALL "CF$ASN".
     CALL "CF$REA" USING GRP, R-DATA, R-DATA-END.
*
* PERFORM UNRELATED PROCESSING.
*******************************
*
     PERFORM DO-SOMETHING-ELSE.
*
* SYNCHRONIZE APPLICATION AND EXECUTOR.
***************************************
*
     CALL "CF$SYN".
*
* PROCESS THE RESULTS OF THE READ.
*
  PERFORM STATUS-CHECK.
```

Pascal:

```
                .
                .
                .
PROCEDURE PX$ASN; EXTERNAL;
PROCEDURE PX$SYN; EXTERNAL;
PROCEDURE DO_SOMETHING_ELSE;
                .
                .
                .
{ THE READ CALL IS TO BE EXECUTED ASYNCHRONOUSLY.
 ================================================}
   PX$ASN;
   PX$REA (GROUP,R_DATA,UB(R_DATA));

{ PERFORM UNRELATED PROCESSING.
 ==============================}

   DO_SOMETHING_ELSE;

{ SYNCHRONIZE APPLICATION AND EXECUTOR.
 ====================================}

   PX$SYN;

{PERFORM STATUS CHECK ON PREVIOUS COMMAND.}

{ PROCESS THE RESULTS OF THE READ.}
```

FORTRAN:

```
C
C   THE READ CALL IS TO BE EXECUTED ASYNCHRONOUSLY.
C
     CALL FFASN
     CALL FFREA (GNAME,RDATA(1),RDATA(END))
C
C   PERFORM UNRELATED PROCESSING.
C
     CALL UNRELATE
C
C   SYNCHRONIZE APPLICATION AND EXECUTOR.
C
     CALL FFSYN
C
C  PERFORM STATUS CHECK ON PREVIOUS COMMAND.
C
C   PROCESS THE RESULTS OF THE READ.
```

5.22.5 Synchronize Program Notes

Your application should check the status block after each Synchronize command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.23 WRITE A GROUP ROUTINE

This routine performs a write operation on the entire group.

5.23.1 Write a Group Calling Sequences

The calling sequences for the Write a Group routine are as follows:

COBOL 3.1:

CALL "CF$WRI" USING < group name>, < write data>, < write data end>.

COBOL 3.2:

CALL "CX$WRI" USING <group name>, <write data>.

Pascal:

PX$WRI (< group name>, < write data>, < write data size>);

FORTRAN:

CALL FFWRI (< group name), < write data>, < write data end>)

5.23.2 Write a Group Parameters

The following list describes the parameters for the Write a Group routine:

< group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group, field, or variable to write. The name must be left-justified and blank-filled on the right within this item.

< write data> is an item (01-level item in COBOL 3.1) containing the data to write into the elemental members of the specified group.

< write data end> is an item (01-level item in COBOL 3.1) indicating the end of the write data area.

< write data size> in Pascal is the size of the write data item.

5.23.3 Write a Group Results

Status is posted. The data is written into the group's elemental members, giving them new values.

5.23.4 Write a Group Examples

The following examples demonstrate the Write a Group routine:

COBOL 3.1:

```
          .
          .
          .
01  GROUP-NAME PIC X(6) VALUE IS 'GROUP1'.
01  DATA-BEGIN PIC X(80).
01  DATA-END PIC X(2)
          .
          .
          .
    CALL "CF$WRI" USING GROUP-NAME,
                        DATA-BEGIN, DATA-END.
```

Pascal:

```
              .
              .
              .
VAR GROUP:C$6;
    DATA_AREA:PACKED ARRAY[1..48] OF CHAR;
              .
              .
PROCEDURE PX$WRI (GRPNAM:C$6;
                  VAR D_AREA:PACKED ARRAY[1..?] OF CHAR;
                      DSIZE:INTEGER);
                  EXTERNAL;
              .
              .
              .
GROUP:='GROUP1';
PX$WRI (GROUP,DATA_AREA,UB(DATA_AREA));
```

FORTRAN:

```
CHARACTER GNAME*6,WDATA*2(40)
              .
              .
              .
GNAME='GROUP1'
              .
              .
              .
CALL FFWRI (GNAME,WDATA(1),WDATA(40))
```

5.23.5 Write a Group Program Notes

Your application should check the status block after each Write a Group command. If an error occurs, your application should report the error to the user in order to identify the problem.

5.24 WRITE INDEXED ROUTINE

This routine enables the user to specify the index parameters to cause a selective write to a portion of a group. These parameters are not altered by the routine. For information regarding group indexing, see paragraph 1.4.3.

5.24.1 Write Indexed Calling Sequences

The calling sequences for the Write Indexed routine are as follows:

COBOL 3.1:

```
CALL "CF$WX" USING <group name>, <write data>, <write data end>, <index-1>,
                   <index-2>, <count>, <cursor position>.
```

COBOL 3.2:

```
CALL "CX$WX" USING <group name>, <write data>, <index-1>, <index-2>, <count>,
                   <cursor position>.
```

Pascal:

```
PX$WX (<group name>, <write data>, <write data size>, <index-1>, <index-2>,
       <count>, <cursor position>);
```

FORTRAN:

```
      CALL FFWX (<group name>, <write data>, <write data end>, <index-1>, <index-2>,
     1           <count>, <cursor position>)
```

5.24.2 Write Indexed Parameters

The following list describes the parameters for the Write Indexed routine:

<group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to write.

<write data> is an item (01-level item in COBOL 3.1) containing the data to write to the named group.

<write data end> is an item (01-level item in COBOL 3.1) indicating the end of the write data area.

<write data size> in Pascal is an item indicating the size of the write data area.

<index-1> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the first-level index of the group's first elemental member to write.

< index-2> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the second-level index of the group's first elemental member to write.

< count> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the number of elemental members to write.

< cursor position> is ignored if specified.

5.24.3 Write Indexed Results

Status is posted. The write data is written into the group's elemental members.

5.24.4 Write Indexed Examples

The following examples demonstrate the Write Indexed routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME        PIC X(6) VALUE IS "NAME00".
01   DATA-BEGIN        PIC X(80).
01   DATA-END          PIC X(2).
01   INDEX-1           PIC 999.
01   INDEX-2           PIC 999.
01   COUNT             PIC 999.
01   CURSOR-POSITION   PIC 999.
          .
          .
          .
     MOVE 05 TO INDEX-1.
     MOVE 08 TO INDEX-2.
     MOVE 03 TO COUNT.   MOVE 02 TO CURSOR-POSITION.
     CALL "CF$WX" USING GROUP-NAME,
                        DATA-BEGIN, DATA-END,
                        INDEX-1, INDEX-2, COUNT.
```

Pascal:

```
              .
              .
              .
VAR GROUP:C$6;
    W_DATA:PACKED ARRAY[1..10] OF CHAR;
    X_1,X_2,COUNT,CR_POS:C$3;
              .
              .
              .
PROCEDURE PX$WX (GRP:C$6;
                 VAR WRTDAT:PACKED ARRAY[1..?] OF CHAR;
                     WRTSIZ:INTEGER;
                 VAR X1,X2,CNT,CPOS:C$3); EXTERNAL;
              .
              .
              .
{Write the third and fourth element of GROUP9.}
GROUP:='GROUP9';
X_1:='003'; X_2:='000'; CNT:='002'; CR_POS:='000';
PX$WX (GROUP,W_DATA,UB(W_DATA),X_1,X_2,COUNT,CR_POS);
```

FORTRAN:

```
      CHARACTER GNAME*6,WDATA*2(40)
      CHARACTER*3 INDEX1,INDEX2,COUNT,CURPOS

                .
                .
                .
C Write the third and fourth element of GROUP9.
      GNAME='GROUP9'
      INDEX1='003'
      INDEX2='000'
      COUNT='002'
      CURPOS='000'
                .
                .
                .
      CALL FFWX (GNAME,WDATA(1),WDATA(40),INDEX1,INDEX2,COUNT,CURPOS)
```

5.24.5 Write Indexed Program Notes

Your application should check the status block after each Write Indexed command. If an error occurs, your application should report the error to the user in order to identify the problem.

5.25 WRITE INDEXED WITH REPLY AND CURSOR RETURN ROUTINE

This routine is the same as Write Indexed With Reply except that the indexing parameters are modified by the routine to indicate the field and column the cursor was in when the Read was terminated. If the Read was terminated by an armed event key or the Enter key, the indexing parameters indicate the final position of the cursor.

For more information regarding group indexing, refer to paragraph 1.4.3.

5.25.1 Write Indexed With Reply and Cursor Return Calling Sequences

The calling sequences for the Write Indexed With Reply and Cursor Return routine are as follows:

COBOL 3.1:

```
CALL "CF$WRC" USING <group name> <write data>, <write data end>, <read data>,
                    <read data end>, <index-1>, <index-2>, <count>,
                    <cursor position>.
```

COBOL 3.2:

```
CALL "CX$WRC" USING <group name>, <write data>, <read data>, <index-1>,
                    <index-2>, <count>, <cursor position>.
```

Pascal:

```
PX$WRC (<group name>, <write data>, <write data size>, <read data>,
        <read data size>, <index-1>, <index-2>, <count>, <cursor position>);
```

FORTRAN:

```
      CALL FFWRC (<group name>, <write data>, <write data end>, <read data>,
     1            <read data end>, <index-1>, <index-2>, <count>, <cursor position>)
```

5.25.2 Write Indexed With Reply and Cursor Return Parameters

The following list describes the parameters for the Write Indexed With Reply and Cursor Return routine:

<group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to write and read.

<write data> is an item (01-level item in COBOL 3.1) containing the data to write to the named group.

<write data end> is an item (01-level item in COBOL 3.1) indicating the end of the write data area.

<write data size> in Pascal is an item indicating the size of the write data area.

<read data> is an item (01-level item in COBOL 3.1) into which the group's elemental members' values are placed.

< read data end> is an item (01-level item in COBOL 3.1) indicating the end of the read buffer.

< index-1> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the first-level index of the group's first elemental member to write/read. The first-level index of the field containing the cursor is returned in this parameter.

< index-2> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the second-level index of the group's first elemental member to write/read. The second-level index of the field containing the cursor is returned in this parameter.

< count> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the number of elemental members to write/read. The elemental number of the field containing the cursor is returned in this parameter.

< cursor position> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the character to start writing/reading within the indexed field. The position of the cursor within the current field is returned in this parameter.

5.25.3 Write Indexed With Reply and Cursor Return Results

Status is posted. The data from the group's elemental members is placed in < read data>. The current cursor position is returned in the indexing parameters.

5.25.4 Write Indexed With Reply and Cursor Return Examples

The following examples demonstrate the Write Indexed With Reply and Cursor Return routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME       PIC X(6) VALUE IS "NAMEXX".
01   WRITE-DATA       PIC X(80).
01   WRITE-DATA-END   PIC X(2).
01   READ-DATA        PIC X(80).
01   READ-DATA-END    PIC X(2)
01   INDEX-1          PIC 999.
01   INDEX-2          PIC 999.
01   COUNT            PIC 999.
01   CURSOR           PIC 999.
          .
          .
          .
     MOVE 02 TO INDEX-1.
     MOVE 06 TO INDEX-2.
     MOVE 04 TO COUNT.
     CALL "CF$WRC" USING GROUP-NAME,
                         WRITE-DATA, WRITE-DATA-END,
                         READ-DATA, READ-DATA-END,
                         INDEX-1, INDEX-2, COUNT, CURSOR.
```

Pascal:

```
            .
            .
            .
VAR GROUP:C$6;
    W_DATA,R_DATA:PACKED ARRAY[1..18] OF CHAR;
    X_1,X_2,COUNT,CR_POS:C$3;
            .
            .
            .
PROCEDURE PX$WRC (GRP:C$6;
                  VAR WRTDAT:PACKED ARRAY[1..?] OF CHAR;
                      WRTSIZ:INTEGER;
                  VAR READAT:PACKED ARRAY[1..?] OF CHAR;
                      REASIZ:INTEGER;
                  VAR X1,X2,CNT,CPOS:C$3); EXTERNAL;
            .
            .
            .
{Write and read five elements beginning in row 2, column 4,
beginning with the sixth character of that element.}
GROUP:='ARY5  ';
X_1:='002'; X_2:='004'; CNT:='005'; CR_POS:='006';
PX$WRC(GROUP,W_DATA,UB(W_DATA),R_DATA,UB(R_DATA),
      X_1,X_2,COUNT,CR_POS);
```

FORTRAN:

```
      CHARACTER GNAME*6,WDATA*2(40)
      CHARACTER RDATA*2(40)
      CHARACTER*3 INDEX1,INDEX2,COUNT,CURPOS
                .
                .
                .
C     Write and read five elements beginning in row 2, column 4,
C     beginning with the sixth character of that element.

      GNAME='ARY5  '
      INDEX1='002'
      INDEX2='004'
      COUNT='005'
      CURPOS='006'
                .
                .
                .
      CALL FFWRC (GNAME,WDATA(1),WDATA(40),RDATA(1),RDATA(40),
     1            INDEX1,INDEX2,COUNT,CURPOS)
```

5.25.5 Write Indexed With Reply and Cursor Return Program Notes

Your application should check the status block after each Write Indexed With Reply and Cursor Return command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.26 WRITE INDEXED WITH REPLY ROUTINE

This routine writes values to a portion of a group and reads values back from the same fields. There are two data buffers, one for writing and one for reading. The same buffer can be used for both. The indexing parameters allow the user to select a portion of the group to write and read. These parameters are not altered by this routine. For information regarding group indexing, refer to paragraph 1.4.3.

5.26.1 Write Indexed With Reply Calling Sequences

The calling sequences for the Write Indexed With Reply routine are as follows:

COBOL 3.1:

```
CALL "CF$WXR" USING <group name>, <write data>, <write data end>, <read data>,
                    <read data end>, <index-1>, <index-2>, <count>,
                    <cursor position>.
```

COBOL 3.2:

```
CALL "CX$WXR" USING <group name>, <write data>, <read data>, <index-1>,
                    <index-2>, <count>, <cursor position>.
```

Pascal:

```
PX$WXR (<group name>, <write data>, <write data size>, <read data>,
        <read data size>, <index-1>, <index-2>, <count>, <cursor position>);
```

FORTRAN:

```
      CALL FFWXR (< group name>, < write data>, < write data end>, < read data>,
     1            < read data end>, < index-1>, < index-2>, < count>,
     1            < cursor position>)
```

5.26.2 Write Indexed With Reply Parameters

The following list describes the parameters for the Write Indexed With Reply routine:

<group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group/field/variable to read.

<write data> is an item (01-level item in COBOL 3.1) containing the data to write into the group's elemental members.

<write data end> is an item (01-level item in COBOL 3.1) indicating the end of the write data area.

<write data size> in Pascal is an item indicating the size of the write data area.

< read data> is an item (01-level item in COBOL 3.1) into which the group's elemental members' values are placed.

< read data end> is an item (01-level item in COBOL 3.1) indicating the end of the read buffer.

< read data size> in Pascal is an item indicating the size of the read buffer.

< index-1> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the first-level index of the group's first elemental member to read.

< index-2> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the second-level index of the group's first elemental member to read.

< count> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the number of elemental members to read.

< cursor position> is a three-byte item (01-level numeric item in COBOL 3.1) specifying the character to start reading within the indexed field.

5.26.3 Write Indexed With Reply Results

Status is posted. The write data is placed in the group's elemental members, and the members' new values are placed in the read buffer.

5.26.4 Write Indexed With Reply Examples

The following examples demonstrate the Write Indexed With Reply routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME       PIC X(6) VALUE IS "NAMEXX".
01   WRITE-DATA       PIC X(80).
01   WRITE-DATA-END   PIC X(2).
01   READ-DATA        PIC X(80).
01   READ-DATA-END    PIC X(2).
01   INDEX-1          PIC 999.
01   INDEX-2          PIC 999.
01   COUNT            PIC 999.
01   CURSOR           PIC 999.
          .
          .
          .
     MOVE 03 TO INDEX-1.
     MOVE 09 TO INDEX-2.
     MOVE 06 TO COUNT, CURSOR.
     CALL "CF$WXR" USING GROUP-NAME,
                         WRITE-DATA, WRITE-DATA-END,
                         READ-DATA, READ-DATA-END,
                         INDEX-1, INDEX-2, COUNT, CURSOR.
```

Pascal:

```
            .
            .
            .
VAR GROUP:C$6;
    W_DATA,R_DATA:PACKED ARRAY[1..20] OF CHAR;
    X_1,X_2,COUNT,CR_POS:C$3;
            .
            .
            .
PROCEDURE PX$WXR (GRP:C$6;
                  VAR WRTDAT:PACKED ARRAY[1..?] OF CHAR;
                      WRTSIZ:INTEGER;
                  VAR READAT:PACKED ARRAY[1..?] OF CHAR;
                      REASIZ:INTEGER;
                  VAR X1,X2,CNT,CPOS:C$3); EXTERNAL;
            .
            .
            .
{Write and read four fields beginning with the fifth.}
GROUP:='GROUP7';
X_1:='005'; X_2:='000'; CNT:='004'; CR_POS:='000';
PX$WXR(GROUP,W_DATA,UB(W_DATA),R_DATA,UB(R_DATA),
                  X_1,X_2,COUNT,CR_POS);
```

FORTRAN:

```
 CHARACTER GROUP*6,WDATA*2(40)
 CHARACTER RDATA*2(40)
 CHARACTER*3 INDEX1,INDEX2,COUNT,CURPOS
      .
      .
      .
 GROUP='GROUP7'
 INDEX1='005'
 INDEX2='000'
 COUNT='004'
 CURPOS='000'
 CALL FFWXR (GROUP,WDATA(1),WDATA(40),RDATA(1),RDATA(40),
1             INDEX1,INDEX2,COUNT,CURPOS)
```

5.26.5 Write Indexed With Reply Program Notes

Your application should check the status block after each Write Indexed With Reply command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.27 WRITE MESSAGE ROUTINE

This routine writes a character string to the message area on the screen. For a VDT, the message area is the twenty-fourth line of the screen. The terminal operator must respond to this message, as with TIFORM error messages, by pressing the Return key or the Enter key to indicate that the user sees the message. This call provides the application with a method to issue its own error messages in a way consistent with that of TIFORM.

Note that the maximum length message that can be written is 78 characters, not 80.

5.27.1 Write Message Calling Sequences

The calling sequences for the Write Message routine are as follows:

COBOL 3.1:

```
CALL "CF$WM" USING < message> , < message end> .
```

COBOL 3.2:

```
CALL "CX$WM" USING < message> .
```

Pascal:

```
PX$WM (< message> , < message size> );
```

FORTRAN:

```
CALL FFWM (< message> , < message end> )
```

5.27.2 Write Message Parameters

The following list describes the parameters for the Write Message routine:

< message> is the item (01-level item in COBOL 3.1) containing the text to display.

< message end> is an item (01-level item in COBOL 3.1) indicating the end of < message> .

< message size> in Pascal is the item containing the size of < message> .

5.27.3 Write Message Results

The message is written on line 24 of the display and status is posted.

5.27.4 Write Message Examples

The following examples demonstrate the Write Message routine:

COBOL 3.1:

```
          .
          .
          .
01  MESSAGE  PIC X(78) VALUE "WHOOPS-A-DAISY...".
01  MSG-END  PIC XX.
          .
          .
          .
    CALL "CF$WM" USING MESSAGE, MSG-END.
```

Pascal:

```
             .
             .
             .
VAR MESSAGE:PACKED ARRAY[1..18] OF CHAR;
             .
             .
             .
PROCEDURE PX$WM (VAR MSG:PACKED ARRAY[1..?] OF CHAR;
                     MSGSIZ:INTEGER); EXTERNAL
             .
             .
             .
MESSAGE:='This is a message.';
PX$WM (MESSAGE,UB(MESSAGE));
```

FORTRAN:

```
CHARACTER MESSAGE*2(9)
DATA MESSAGE/'This is a message.'/
             .
             .
             .
CALL FFWM (MESSAGE(1),MESSAGE(9))
```

5.27.5 Write Message Program Notes

Your application should check the status block after each Write Message call. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

5.28 WRITE WITH REPLY ROUTINE

This routine writes values to a group and reads values back from the same fields. There are two data buffer parameters, one for writing and one for reading, but the same buffer can be used for both functions.

5.28.1 Write With Reply Calling Sequences

The calling sequences for the Write With Reply routine are as follows:

COBOL 3.1:

```
CALL "CF$WWR" USING < group name> , < write data> , < write data end> , < read data> ,
                    < read data end> .
```

COBOL 3.2:

```
CALL "CX$WWR" USING < group name> , < write data> , < read data> .
```

Pascal:

```
PX$WWR (< group name> , < write data> , < write data size> , < read data> ,
        < read data size> );
```

FORTRAN:

```
      CALL FFWWR (< group name> , < write data> , < write data end> , < read data> ,
     1                < read data end> )
```

5.28.2 Write With Reply Parameters

The following list describes the parameters for the Write With Reply routine:

< group name> is a six-byte item (01-level item in COBOL 3.1) containing the name of the group, field, or variable to write. The name must be left-justified and blank-filled on the right within this item.

< write data> is an item (01-level item in COBOL 3.1) containing the data to write into the elemental members of the specified group.

< write data end> is an item (01-level item in COBOL 3.1) indicating the end of the write data area.

< write data size> in Pascal is an item indicating the size of the write data area.

< read data> is an item (01-level item in COBOL 3.1) into which the group's elemental members' values are read.

< read data end> is an item (01-level item in COBOL 3.1) indicating the end of the read data area.

< read data size> in Pascal is an item indicating the size of the read data area.

5.28.3 Write With Reply Results

Status is posted. The write data area is written into the group's elemental members. The new values of the group's elemental members are read into the read data area.

5.28.4 Write With Reply Examples

The following examples demonstrate the Write With Reply routine:

COBOL 3.1:

```
          .
          .
          .
01   GROUP-NAME PIC X(6) VALUE IS "NAME09".
01   WRITE-DATA      PIC X(80).
01   WRITE-DATA-END  PIC X(2).
01   READ-DATA       PIC X(80).
01   READ-DATA-END   PIC X(2).
          .
          .
          .
     CALL "CF$WWR" USING GROUP-NAME,
                         WRITE-DATA, WRITE-DATA-END,
                         READ-DATA, READ-DATA-END.
```

Pascal:

```
               .
               .
               .
VAR GROUP:C$6;
    W_DATA,R_DATA:PACKED ARRAY[1..34] OF CHAR;
               .
               .
               .
PROCEDURE PX$WWR (GRPNAM:C$6;
                  VAR WRTDAT:PACKED ARRAY[1..?] OF CHAR;
                      WRTSIZ:INTEGER;
                  VAR READAT:PACKED ARRAY[1..?] OF CHAR;
                      RDSIZ:INTEGER); EXTERNAL;
               .
               .
               .
GROUP:='GROUP1';
PX$WWR (GROUP,W_DATA,UB(W_DATA),R_DATA,UB(R_DATA));
```

FORTRAN:

```
CHARACTER GNAME*6,WDATA*2(40)
CHARACTER RDATA*2(40)
          .
          .
          .
GNAME='GROUP1'
          .
          .
          .
CALL FFWWR (GNAME,WDATA(1),WDATA(40),RDATA(1),RDATA(40))
```

5.28.5 Write With Reply Program Notes

Your application should check the status block after each Write With Reply command. If the status code is nonzero, your application should report the code to the user and take the appropriate end-action.

6

Linking Application Programs That Use TIFORM

6.1 INTRODUCTION

After you compile your application program containing TIFORM high-level language (HLL) routine calls, you are ready to link it with the HLL interface modules. There are two techniques for linking these modules to use the Form Executor. The multitask version allows several tasks to use the Form Executor at the same time. In the linkable version, the Form Executor is linked directly with your application program and resides in the same program file.

6.2 USING MULTITASK TIFORM

An application program uses TIFORM by calling the routines described in Section 5. These routines reside in the TIFORM high-level language (HLL) interface modules, of which there are five. Each HLL interface module implements one unique calling sequence. An application program using TIFORM must be linked with one of these interface modules. The modules define the entry points for the routines described in Section 5 and provide all communication with the TIFORM Form Executor task.

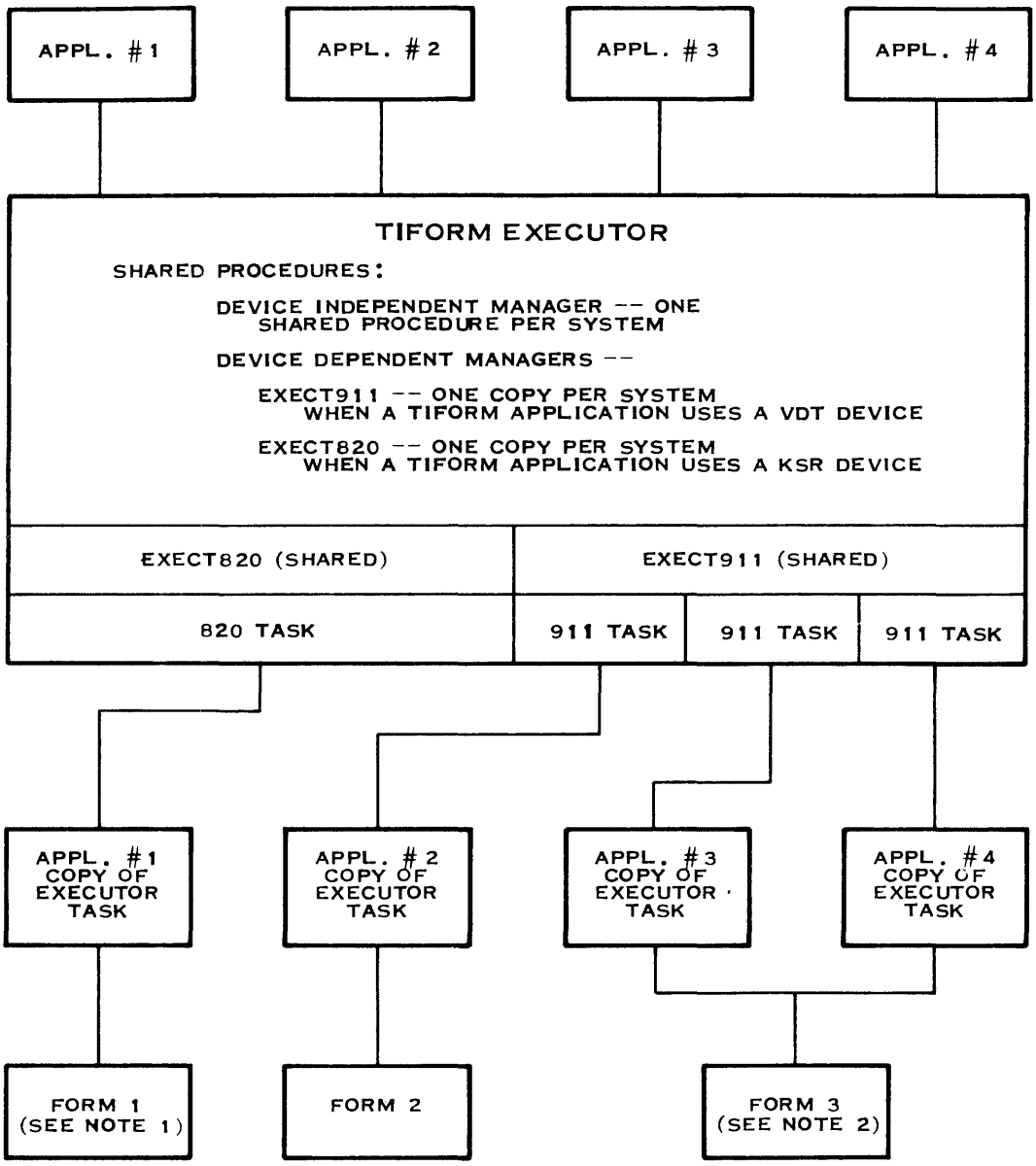
Figure 6-1 shows how several applications can share the Form Executor in a multitasking environment.

There are five interface modules provided with the TIFORM installation: one for FORTRAN, two for COBOL, and two for Pascal. These modules reside in the directory .S$TIFORM.O and have the following names:

.S$TIFORM.O

- .CF$MTASK COBOL 3.1 interface module.
- .CX$MTASK COBOL 3.2 interface module.
- .FF$MTASK FORTRAN interface module.
- .PF$MTASK Pascal external FORTRAN interface module.
- .PX$MTASK Standard Pascal interface module.

You must explicitly include one of these modules in a link edit of the application.



NOTES:

1. A FORM CONSISTS OF A GROUP OF SEGMENTS, SEGMENT MASKS, AND A FORM ROOT. EACH COMPONENT OF THE FORM RESIDES IN A OVERLAY.
2. THIS ILLUSTRATES HOW MORE THAN ONE APPLICATION CAN USE A SINGLE FORM.

2285375

Figure 6-1. Multitask TIFORM

For compatibility with previous releases of TIFORM, the following aliases are defined in .S$TIFORM.O to allow existing link control files to continue to work:

CF$XFACE is an alias of CF$MTASK

FF$XFACE is an alias of FF$MTASK

PF$XFACE is an alias of PF$MTASK

The following five examples show the link control files necessary to link TIFORM with the following:

- COBOL
  - 3.1
  - 3.2
- Pascal
  - Standard Pascal
  - External FORTRAN
- FORTRAN

EXAMPLE 1

```
;
; Link control file of COBOL 3.1 TIFORM example...
;
PROCEDURE RTCOBOL
 INCLUDE .S$SYSLIB.RCBPRC
TASK FORMTSTR
 INCLUDE .S$SYSLIB.RCBTSK
 INCLUDE .S$SYSLIB.RCBMPD
 INCLUDE .EXAMPLE.COBOLOBJ        The COBOL program.
 INCLUDE .S$TIFORM.O.CF$MTASK     Include CF$xxx modules.
END
```

EXAMPLE 2

```
;
; Link control file of COBOL 3.2 TIFORM example...
;
PROCEDURE RTCOBOL
 INCLUDE .S$SYSLIB.RCBPRC
TASK FORMTSTR
 INCLUDE .S$SYSLIB.RCBTSK
 INCLUDE .S$SYSLIB.RCBMPD
 INCLUDE .EXAMPLE.COBOLOBJ       The COBOL program.
 INCLUDE .S$TIFORM.O.CX$MTASK    Include CX$xxx modules.
END
```

EXAMPLE 3

```
;
; Link control file of FORTRAN TIFORM example...
;
LIBRARY .FORT78.OSLOBJ
LIBRARY .FORT78.STLOBJ
LIBRARY .SCI990.S$OBJECT          Pick up SCI S$xxxx modules.
TASK FORTST
 INCLUDE .EXAMPLE.FORTNOBJ        The FORTRAN program.
 INCLUDE .S$TIFORM.O.FF$MTASK     Get FFxxx modules.
 INCLUDE .SCI990.S$OBJECT.S$TCA$  Include enough S$xxxx modules
 INCLUDE .SCI990.S$OBJECT.S$STOP  to override the modules in
 INCLUDE .SCI990.S$OBJECT.S$SETS  .STLOBJ and allow synonym
 INCLUDE .SCI990.S$OBJECT.S$IO    resolution to work.
 INCLUDE .SCI990.S$OBJECT.S$MAPS  *
END
```

EXAMPLE 4

```
;
; Link control file of EXTERNAL FORTRAN
;   Pascal TIFORM example...
;
LIBRARY .TIP.OBJ
TASK EXAMPLE
 INCLUDE (MAIN)
 ALLOCATE
 INCLUDE .EXAMPLE.PASCLOBJ     The Pascal example program.
 INCLUDE .S$TIFORM.O.PF$MTASK  Include PF$xxx routines.
END
```

EXAMPLE 5

```
;
; Link control file of TI Pascal TIFORM example...
;
LIBRARY .TIP.OBJ
 TASK EXAMPLE
 INCLUDE (MAIN)
 ALLOCATE
 INCLUDE .EXAMPLE.PASCLOBJ      The Pascal example program.
 INCLUDE .S$TIFORM.O.PX$MTASK   Include PX$xxx routines.
END
```

6.3 LINKABLE TIFORM EXECUTORS

If you do not want your application system to use a separate task for the TIFORM Executor, you can link the TIFORM Executor directly with some component of the application. The primary advantage of this process is that the ITC/IPC communication used by the standard multitask TIFORM is eliminated when the application and the Executor are linked together. (Under DX10, TIFORM uses the ITC. Under DNOS, TIFORM uses the IPC.) The primary disadvantages are that the Executor uses much of the address space when linked with an application and that a linkable Executor can support only a single terminal type. Figure 6-2 shows the structure of a TIFORM application linked directly with a linkable Executor.

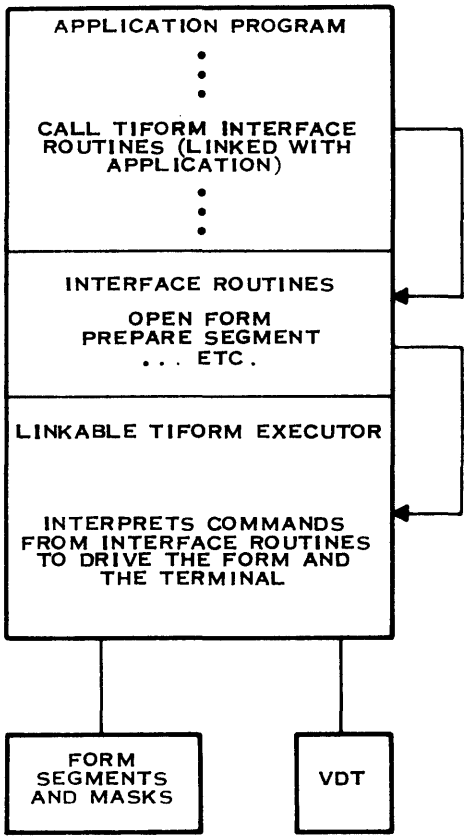
The standard TIFORM Executor and the HLL interface modules in .S$TIFORM.O are not suitable for linking directly with an application. Special modules are needed. Consequently, a separate directory, TIFRMINS.LINKD, is provided on the object release disk. This directory contains the necessary components for building linkable Executors. Due to the size of the directory and the rarity of the need for a linkable Executor, this directory is not copied by the TIFORM installation. It exists only on TIFRMINS.

6.3.1 Building a Linkable Executor

To create a linkable Executor, first install the SCI command BLDLINKD in .S$PROC. The source for this procedure is in the TIFRMINS.LINKD directory. The BLDLINKD command builds the specific linkable Executor requested. It assumes that the synonym @TIFRMINS points to the directory under which the .LINKD directory resides.

The BLDLINKD command prompts for the following information:

```
BUILD A LINKABLE TIFORM EXECUTOR
   LANGUAGE(CF$,CX$,PF$,PX$,FF$): CX$
              TERMINAL(VDT,820): VDT
        LINKED OUTPUT DIRECTORY:
              LISTING DIRECTORY:
```



2285376

Figure 6-2. TIFORM With a Linkable Executor

Each linkable Executor constructed consists of a specific language interface together with support for a specific terminal. The first two prompts request the language interface and the terminal type. The last two prompts request pathnames for the directories into which the object module and link map of the Executor are to be placed. Both files are created using the following file name:

< language type> < terminal type>

For example, a linkable Executor using the COBOL 3.2 interface modules and support for the 911 terminal is given the file name CX$911 within the specified directories.

The BLDLINKD procedure first constructs a link control stream using the Copy/Concatenate (CC) SCI command. It then performs a link edit, placing the linked object module and the link map listing files in the specified directories. Once this link edit is finished, the linkable Executor is ready for use.

6.3.2 Using a Linkable Executor

Linking an application with a linkable Executor differs from the multitask examples shown previously in two respects. First, instead of including an HLL interface module from .S$TIFORM.O, you must include the linkable Executor module built by BLDLINKD. Second, the module NFM$TB (in the directory @TIFRMINS.LINKD) must reside at the end of the application task's address space, so you must explicitly include it last.

For example, assume that a COBOL 3.2/VDT linkable Executor is created and its object module is placed in the directory .S$TIFORM.O. Assuming the synonym TIFRMINS is assigned to the directory containing LINKD, the following example shows the link control stream for this Executor.

EXAMPLE

```
;
; Link control file showing use of a linkable
; COBOL 3.2, VDT linkable executor...
;
PROCEDURE RTCOBOL
 INCLUDE .S$SYSLIB.RCBPRC
TASK APPLICTN
 INCLUDE .S$SYSLIB.RCBTSK
 INCLUDE .S$SYSLIB.RCBMPD
 INCLUDE .EXAMPLE.COBOLOBJ      The COBOL program.
 INCLUDE .S$TIFORM.O.CX$911     Include linkable executor.
 INCLUDE TIFRMINS.LINKD.NFM$TB  Include final module.
END
```

If you link the Form Executor directly with your application, the resultant program contains the version of the Form Executor of the latest release. It does not contain any patches that may have been applied since the release. In order to apply the latest patches to the linkable Executor, *you must save the link map* containing the load address of NFM$TB. The TIFORM Release Information discusses how you apply patches to the linkable Executor.

7

Form Tester Utility

7.1 INTRODUCTION

The Form Tester utility permits the testing of any form without the need of a special program to drive the form. In addition, it provides for the deletion of form segments, segment masks, and roots from a form program file. If you are using the DX10 operating system, you can use the Form Tester and the Intertask Channel Clearer utility to develop your forms. If you are using the DNOS operating system, you only need the Form Tester.

7.2 FORM TESTER

This paragraph explains how to use the Form Tester. You invoke the Form Tester by entering the following SCI command with no parameters:

```
FORMTSTR
```

The basic cycle of interaction between you and the Form Tester is as follows:

1. The Form Tester clears the screen, displays a menu of all the possible TIFORM commands, or activities (Figure 7-1), and asks which command it should issue. Enter the appropriate two-digit number.

2. The Form Tester clears the screen and displays a menu requesting values for all the parameters of the specified activity. Enter the desired value for each parameter, left-justified in its field. The Form Tester does not check the accuracy of the values entered.

3. After you enter the last parameter value, the prompt SURE? appears on the bottom (left side) of the screen. An N response redisplays the menu of commands; the command is not issued. Respond with Y to issue the response.

4. The Form Tester issues the command to the Form Executor. When the Form Executor returns control, the Form Tester positions the cursor in the lower right corner of the screen. Either press the Return key to return to step 1, or enter an activity number. If you enter an activity number, the menu for the selected activity appears.

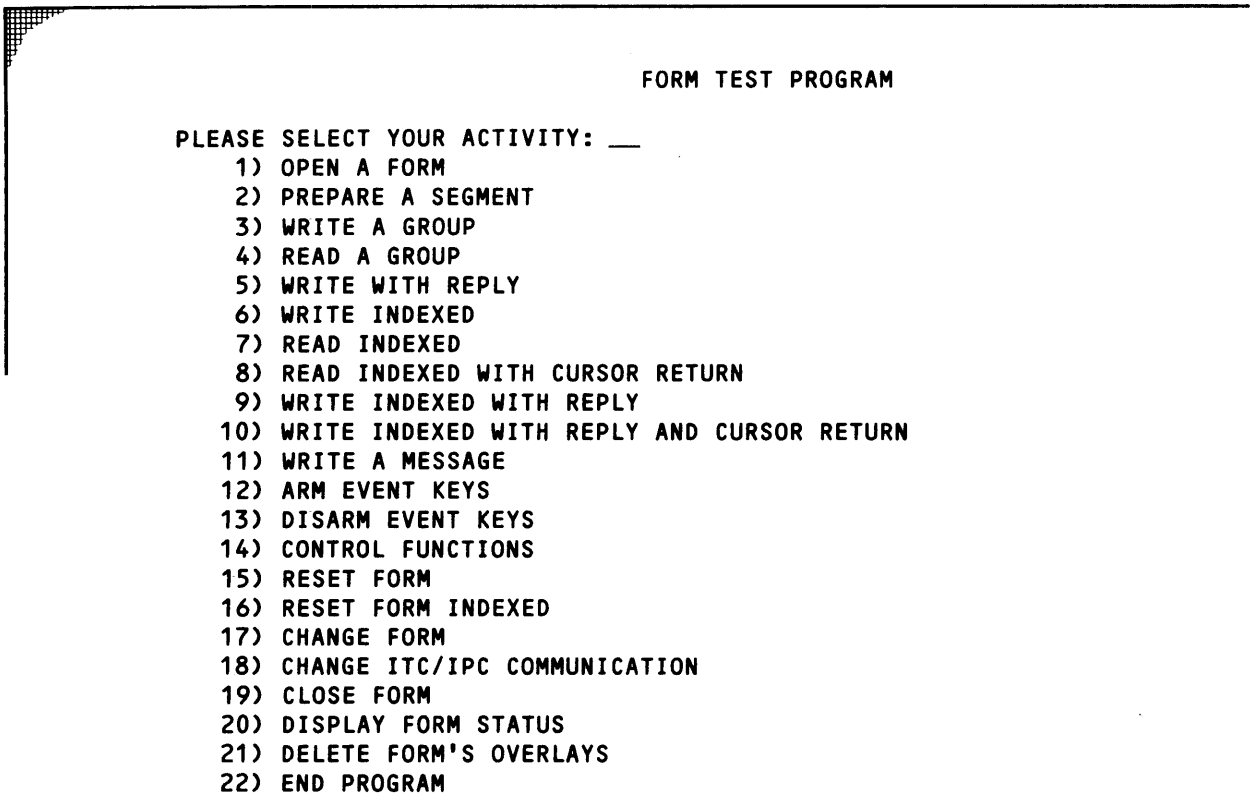


Figure 7-1. Form Tester Activity Selection Menu

The following paragraphs explain each of the Form Tester activities. The first 19 activities of the Activity Selection menu request that specific TIFORM commands be executed. For more detailed explanation of these TIFORM commands, refer to Section 5. The last three activities—activities 20, 21, and 22—request special actions of the Form Tester.

NOTE

When the Form Tester prompts you for a name, such as a segment name, you must enter the name in uppercase characters only. The Form Tester does not map uppercase to lowercase. When executing one of your segments, you can enter lowercase characters if your field definitions accept lowercase.

7.2.1 Open a Form (Activity 1)

This activity opens the specified form. It prompts you for a form name, a program file name, and a terminal name. The form name is the name you specified in your FDL source. This is also the name of the form root overlay in the program file. The program file name identifies where the form overlays reside on disk. You can use a synonym for the program file pathname. The initial value for the TERMINAL NAME prompt is ME. You can enter any valid station ID, such as ST01. If you enter a station ID other than your own (ME), that station must be available. If that station is in use, the Form Tester returns a status code of 57, indicating that it cannot assign a LUNO to the terminal. The activity validates the prompt responses, bids the Form Executor, and passes them to it. You must issue an Open Form command before any other activity, other than activities 21 and 22, can be performed.

7.2.2 Prepare a Segment (Activity 2)

This activity loads a specified segment of the currently open form into the Form Executor. It displays its background mask and all default values of the segment's fields. The activity prompts you for a segment name and then a group name. If you enter a group name, a read is issued for that group in addition to the prepare segment. This results in the execution of the Read and the display of the segment mask. Usually, a prepare segment followed by a separate read clears the segment mask, and the Read takes place on an empty screen. Since a segment name is implicitly a group name, you can prepare and read the entire segment by entering the segment name for both the segment name and group name prompts.

7.2.3 Write a Group (Activity 3)

This activity performs a Write operation for an entire group. It prompts you for a group name and the write data value. It presents four lines of underscores where you can enter the write data value.

7.2.4 Read a Group (Activity 4)

This activity performs a read operation for an entire group. It prompts you for a group name. A message then appears on your screen displaying the data read.

7.2.5 Write With Reply (Activity 5)

This activity writes values to a group and reads values back from the same fields. It prompts you for a group name and the write data value. It presents four lines of underscores where you can enter the write data value. A message then appears on your screen displaying the data read.

7.2.6 Write Indexed (Activity 6)

This activity writes a portion of a group as specified by the index parameters. These parameters are not altered by the activity. The activity first prompts you for a group name and the write data value. It presents four lines of underscores where you can enter the write data value. The next prompt, INDEX 1, asks you for the first-level index of the group's first elemental member to write. The initial value is 000. The INDEX 2 prompt asks you for the second-level index of the group's first elemental member to write. The initial value is 000. The COUNT prompt asks you to specify the number of elemental members to write. The initial value is 000. The CURSOR POSITION prompt asks for a cursor position. The initial value is 000.

7.2.7 Read Indexed (Activity 7)

This activity reads a portion of a group as specified by the index parameters. These parameters are not altered by the activity. The activity first prompts you for a group name. The next prompt, INDEX 1, asks you for the first-level index of the group's first elemental member to read. The initial value is 000. The INDEX 2 prompt asks you for the second-level index of the group's first elemental member to read. The initial value is 000. The COUNT prompt asks you to specify the number of elemental members to read. The initial value is 000. The CURSOR POSITION prompt asks for a cursor position. The initial value is 000. A message then appears on your screen displaying the data read.

7.2.8 Read Indexed With Cursor Return (Activity 8)

This activity is the same as Read Indexed (activity 7) except that the index parameters are altered to indicate which field the cursor was in when screen input was terminated. The index parameters point to the last field read. If the Read was terminated by an armed event key or the Enter key, the index parameters indicate the final position of the cursor. The activity first prompts you for a group name. The next prompt, INDEX 1, asks you for the first-level index of the group's first elemental member to read. The initial value is 000. The INDEX 2 prompt asks you for the second-level index of the group's first elemental member to read. The initial value is 000. The COUNT prompt asks you to specify the number of elemental members to read. The initial value is 000. The CURSOR POSITION prompt asks for a cursor position. The initial value is 000. A message then appears on your screen displaying the data read.

7.2.9 Write Indexed With Reply (Activity 9)

This activity writes values to a portion of a group and reads values back from the same fields. The index parameters allow you to select a portion of the group to write and read. These parameters are not altered by the activity. The activity prompts you for a group name and the write data value. It presents four lines of underscores where you can enter the write data value. The next prompt, INDEX 1, asks you for the first-level index of the group's first elemental member to write. The initial value is 000. The INDEX 2 prompt asks you for the second-level index of the group's first elemental member to write. The initial value is 000. The COUNT prompt asks you to specify the number of elemental members to write. The initial value is 000. The CURSOR POSITION prompt asks for a cursor position. The initial value is 000. A message then appears on your screen displaying the data read.

7.2.10 Write Indexed With Reply and Cursor Return (Activity 10)

This activity is the same as Write Indexed With Reply except that the index parameters are modified by the activity to indicate the field and column the cursor was in when the Read was terminated. If the Read was terminated by an armed event key or the Enter key, the index parameters indicate the final position of the cursor. The activity prompts you for a group name and the write data value. It presents four lines of underscores where you can enter the write data value. The next prompt, INDEX 1, asks you for the first-level index of the group's first elemental member to write. The initial value is 000. The INDEX 2 prompt asks you for the second-level index of the group's first elemental member to write. The initial value is 000. The COUNT prompt asks you to specify the number of elemental members to write. The initial value is 000. The CURSOR POSITION prompt asks for a cursor position. The initial value is 000. A message then appears on your screen displaying the data read.

7.2.11 Write a Message (Activity 11)

This activity writes a character string to the message area on the screen. It provides one line of underscores where you can enter the message text. The maximum length of the message that can be written is 78 characters.

7.2.12 Arm Event Keys (Activity 12)

This activity allows you to arm event keys. Function keys are the only event keys that you can arm. The function keys specified by this activity completely replace whatever function keys were armed prior to the call. The activity prompts you for the first key to arm and then asks you if you want to abort the arming of that key. It continues prompting until you enter 15 keys or press Return without entering a value.

7.2.13 Disarm Event Keys (Activity 13)

This activity disarms all function keys. It is equivalent to an Arm Event Keys call with no keys specified. There are no prompts other than the standard SURE? prompt.

7.2.14 Control Functions (Activity 14)

This activity specifies various control functions to the Form Executor. The activity prompts you for the code specifying the control function. It presents a list of the various control functions and their corresponding codes.

7.2.15 Reset Form (Activity 15)

This activity reinitializes all fields in the specified group to their default values. It prompts you for a group name.

7.2.16 Reset Form Indexed (Activity 16)

This activity is the same as Reset Form (activity 15) except that you can use the index and count parameters to specify a portion of a group. The activity first prompts you for a group name. The next prompt, INDEX 1, asks you for the first-level index of the group's first elemental member to write. The initial value is 000. The INDEX 2 prompt asks you for the second-level index of the group's first elemental member to write. The initial value is 000. The COUNT prompt asks you to specify the number of elemental members to write. The initial value is 000. The CURSOR POSITION prompt asks for a cursor position. The initial value is 000.

7.2.17 Change Form (Activity 17)

This activity allows you to change a form, the form program file, and/or the terminal/file. If you enter a Y under CHANGE for FORM, underscores prompting for a form root overlay number, under OVERLAY/LUNO, are displayed. If you do not enter an overlay number, underscores prompting for the form name are displayed. If you enter a Y for PROGRAM FILE, underscores prompting for a logical unit number (LUNO) are displayed. If you do not provide a LUNO, underscores prompting for a pathname are displayed. Similarly, for the TERMINAL/FILE, underscores prompting for a LUNO or pathname are displayed. You can also enter a terminal name (for example, ST07) as input. There is no prompting for a parameter value if the response is N for any parameter, and the existing value for that parameter is used when the Change Form command is issued.

7.2.18 Change ITC/IPC Communication (Activity 18)

This activity disconnects the current Executor and connects another Executor. A form must be open when you use this activity. The activity prompts you for the run ID of the Executor you want to use. You should specify the run ID in decimal rather than in hexadecimal.

7.2.19 Close Form (Activity 19)

This activity terminates form processing. To simplify error handling, you can issue a close even if the form was not successfully opened. The only prompt that appears is the standard SURE? prompt.

7.2.20 Display Form Status (Activity 20)

This activity displays the status of the current Form Tester session. Figure 7-2 shows the format of this display. You can either press the Return key to return to the selection menu or enter an activity number.

```
STATUS BLOCK CONTENTS...     SESSION STATUS...   DISCONNECTED FROM EXEC

    FORM STATUS - 00                FORM NAME OPEN: _____
      OS STATUS - 00           CURRENT SEGMENT NAME: _____
      EVENT KEY - 00                LAST GROUP NAME: _____
        INDEX 1 - 000                         INDEX 1: 000    ITEM COUNT: 000
        INDEX 2 - 000                         INDEX 2: 000    CRSR-POSTN: 000
     ITEM COUNT - 000               CONTROL MODES: 1=OFF 2=OFF 3=OFF 4=OFF
     CRSR-POSTN - 000                              5=OFF 6=OFF 7=OFF 8=OFF
    TEXT LENGTH -   0                              9=OFF10=OFF11=OFF12=OFF
                                          EVENT KEYS: ____________________
                                                      ____________________

DATA READ WAS -   ____________________________________________________
                  ____________________________________________________
                  ____________________________________________________
                  ____________________________________________________

DATA WRITTEN WAS - ___________________________________________________
                   ___________________________________________________
                   ___________________________________________________
                   ___________________________________________________

PROGRAM FILE NAME: ___________________________________________

    TERMINAL NAME: ST00

    NOTE: Underscores represent positions where data is displayed.
          They are not actually part of the display.
```

Figure 7-2. Form Tester Status Display

7.2.21 Delete Form's Overlays (Activity 21)

This activity allows you to delete segments, segment masks, and roots from a form program file. A form must not be open when you use this activity. When you enter a form name, all segments and segment masks of that form are listed on a form that might be several pages long. You can protect an element from deletion by changing its associated Y to N. You can use the Command key to abort this operation during the first two screens involved in the deletion process. Press Enter or Return for each list of elements to begin the deletion process. When the process is complete, a list of the elements is displayed. An error code field is associated with each element. If no error code is displayed, the element is deleted successfully. If an error code is displayed for an element, that element is not deleted. The first two digits of the error code specify the SVC on which the error occurred; the last two digits specify the error. To determine the reason the element is not deleted, consult either the *DX10 Operating System Error Reporting and Recovery Manual (Volume VI)* or the *DNOS Messages and Codes Reference Manual*, depending on the operating system you are using.

7.2.22 End Program (Activity 22)

This activity terminates the Form Tester program.

7.3 DX10 INTERTASK CHANNEL CLEARER

The Form Executor communicates with an application program via the DX10 Intertask Communication (ITC) facility. ITC is a standard DX10 service that permits the establishment of communication channels and the exchange of data between two tasks.

Occasionally, a message sent via ITC gets left in a channel. This occurs particularly during application development. Such unreceived messages can eventually congest the DX10 ITC buffer space, inhibiting any further communication between the application and the Form Executor.

Congested ITC channels are detected by retry time-out logic in the message-sending routines of the HLL interface modules and the Form Executor. If the HLL interface routines cannot send a message, a form status of 21 (DX10 status of > FF) is returned to the application. If the Form Executor cannot send a message, the TIFORM error 98 is declared, causing the HLL interface routines to return a form status of 24 to the application.

There are two ways to clear congested ITC channels. Performing an initial program load (IPL) for DX10 clears the ITC channels. If performing an IPL is not practical, you can use the Intertask Channel Clearer utility. Its effect is similar to performing an IPL for DX10 in that all messages are cleared from all ITC channels.

To execute this program, enter the following DX10 command:

```
XT PROGFILE=.S$TIFORM.PROG, TASK=CLEAR
```

NOTE

Because this utility clears all messages from all ITC channels, take care to ensure that no other processes that use ITC are running before executing this utility. Sort/Merge uses the ITC queues, as do several of the DX10 terminal emulation packages (for example, 3270 emulation and some modes of 3780/2780 emulation).

Appendix A

Keycap Cross-Reference

Generic keycap names that apply to all terminals are used for keys on keyboards throughout this manual. This appendix contains specific keyboard information to help you identify individual keys on any supported terminal. For instance, every terminal has an Attention key, but not all Attention keys look alike or have the same position on the keyboard. You can use the terminal information in this appendix to find the Attention key on any terminal.

The terminals supported are the 931 VDT, 911 VDT, 915 VDT, 940 EVT, the Business System terminal, and hard-copy terminals (including teleprinter devices). The 820 KSR has been used as a typical hard-copy terminal. The 915 VDT keyboard information is the same as that for the 911 VDT except where noted in the tables.

Appendix A contains three tables and keyboard drawings of the supported terminals.

Table A-1 lists the generic keycap names alphabetically and provides illustrations of the corresponding keycaps on each of the currently supported keyboards. When you need to press two keys to obtain a function, both keys are shown in the table. For example, on the 940 EVT the Attention key function is activated by pressing and holding down the Shift key while pressing the key labeled PREV FORM NEXT. Table A-1 shows the generic keycap name as Attention, and a corresponding illustration shows a key labeled SHIFT above a key named PREV FORM NEXT.

Function keys, such as F1, F2, and so on, are considered to be already generic and do not need further definition. However, a function key becomes generic when it does not appear on a certain keyboard but has an alternate key sequence. For that reason, the function keys are included in the table.

Multiple key sequences and simultaneous keystrokes can also be described in generic keycap names that are applicable to all terminals. For example, you use a multiple key sequence and simultaneous keystrokes with the log-on function. You log on by *pressing the Attention key, then holding down the Shift key while you press the exclamation (!) key.* The same information in a table appears as *Attention/(Shift)!*.

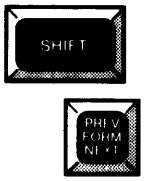
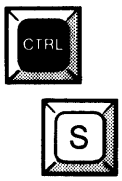
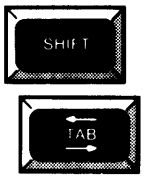
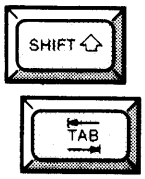
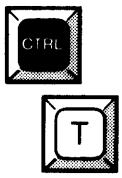
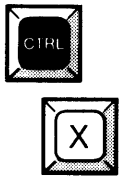
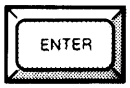
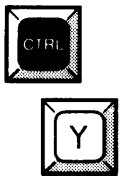
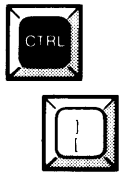
Table A-2 shows some frequently used multiple key sequences.

Table A-3 lists the generic names for 911 keycap designations used in previous manuals. You can use this table to translate existing documentation into generic keycap documentation.

Figures A-1 through A-5 show diagrams of the 911 VDT, 915 VDT, 940 EVT, 931 VDT, and Business System terminal, respectively. Figure A-6 shows a diagram of the 820 KSR.

2274834 (1/14)

Table A-1. Generic Keycap Names

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Alternate Mode	None				None
Attention[2]					
Back Tab	None			None	
Command[2]					
Control					
Delete Character					None
Enter					
Erase Field					

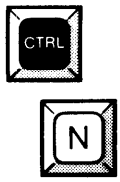
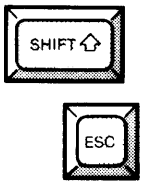
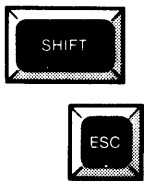
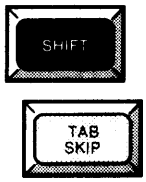
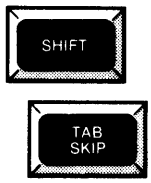
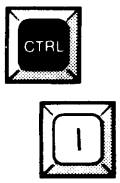
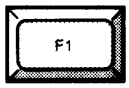
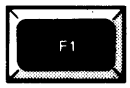
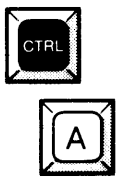
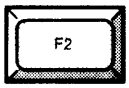
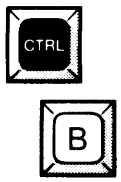
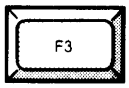
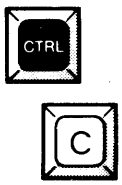
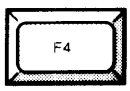
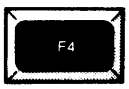
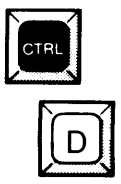
Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

[2]On a 915 VDT the Command Key has the label F9 and the Attention Key has the label F10.

2284734 (2/14)

Table A-1. Generic Keycap Names (Continued)

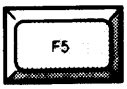
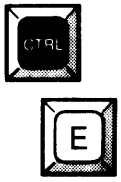
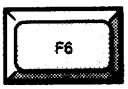
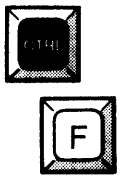
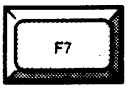
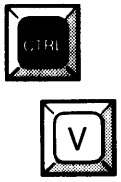
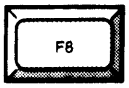
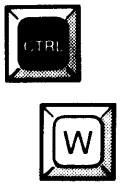
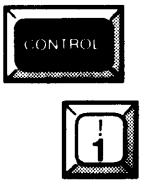
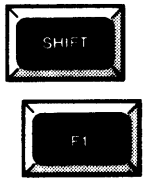
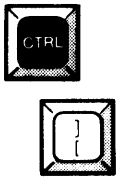
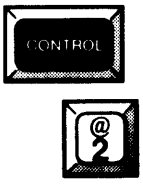
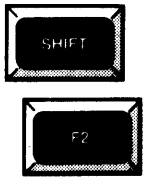
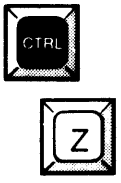
Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Erase Input					
Exit					
Forward Tab					
F1					
F2					
F3					
F4					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (3/14)

Table A-1. Generic Keycap Names (Continued)

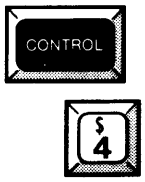
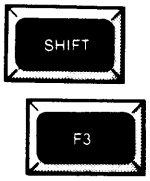
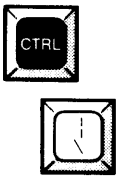
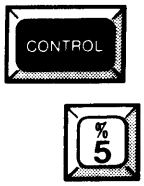
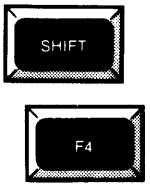
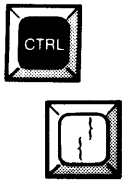
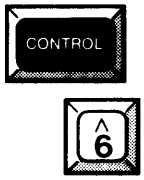
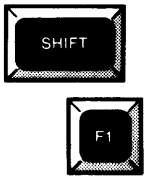
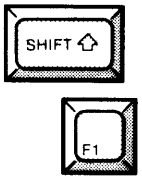
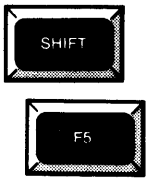
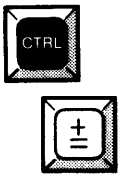
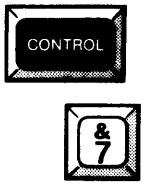
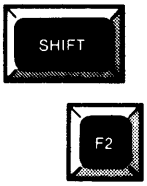
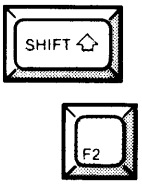
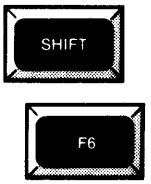
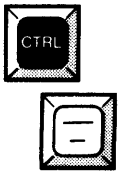
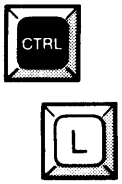
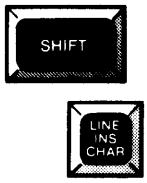
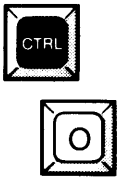
Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
F5					
F6					
F7					
F8					
F9					
F10					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (4/14)

Table A-1. Generic Keycap Names (Continued)

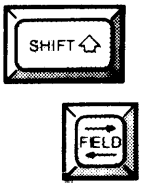
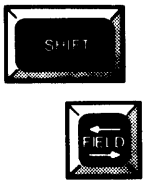
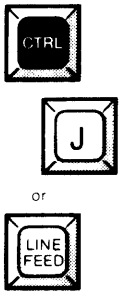
Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
F11					
F12					
F13					
F14					
Home					
Initialize Input					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (5/14)

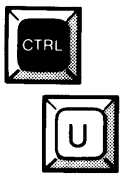
Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Insert Character					None
Next Character	 or 				None
Next Field					None
Next Line					
Previous Character	 or 				None
Previous Field					None

Notes:
[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

2284734 (6/14)

Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820[1] KSR
Previous Line					
Print					None
Repeat		See Note 3	See Note 3	See Note 3	None
Return					
Shift					
Skip					None
Uppercase Lock					

Notes:

[1]The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

[3]The keyboard is typamatic, and no repeat key is needed.

2284734 (7/14)

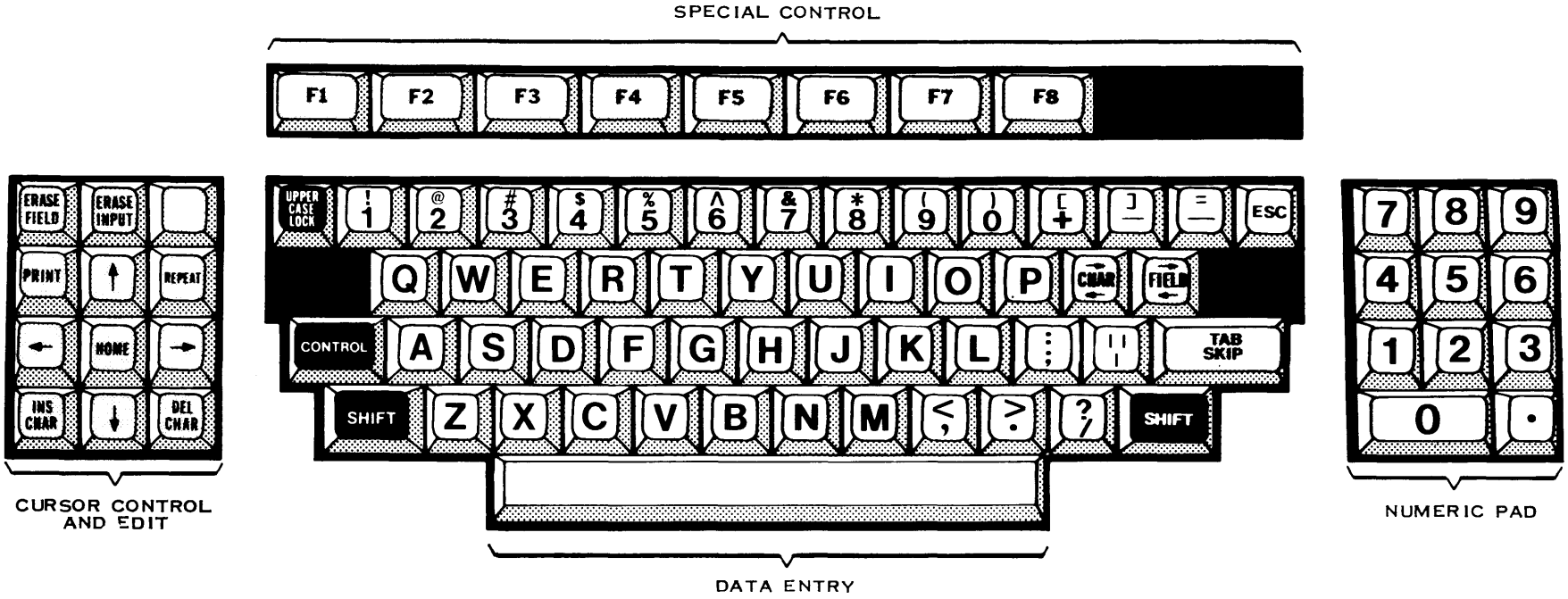
Table A-2. Frequently Used Key Sequences

Function	Key Sequence
Log-on	Attention/(Shift)!
Hard-break	Attention/(Control)x
Hold	Attention
Resume	Any key

Table A-3. 911 Keycap Name Equivalents

911 Phrase	Generic Name
Blank gray	Initialize Input
Blank orange	Attention
Down arrow	Next Line
Escape	Exit
Left arrow	Previous Character
Right arrow	Next Character
Up arrow	Previous Line

2284734 (8/14)



2284734 (9/14)

Figure A-1. 911 VDT Standard Keyboard Layout

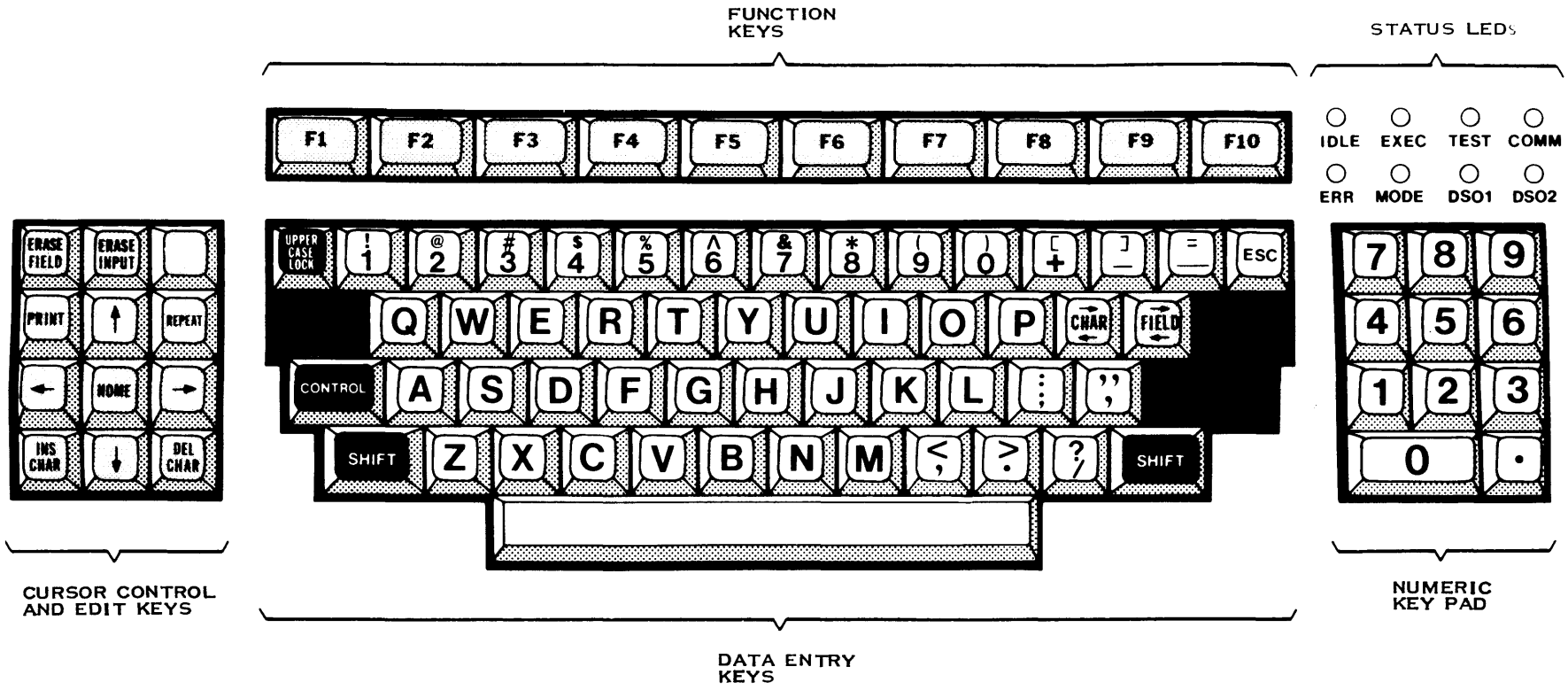
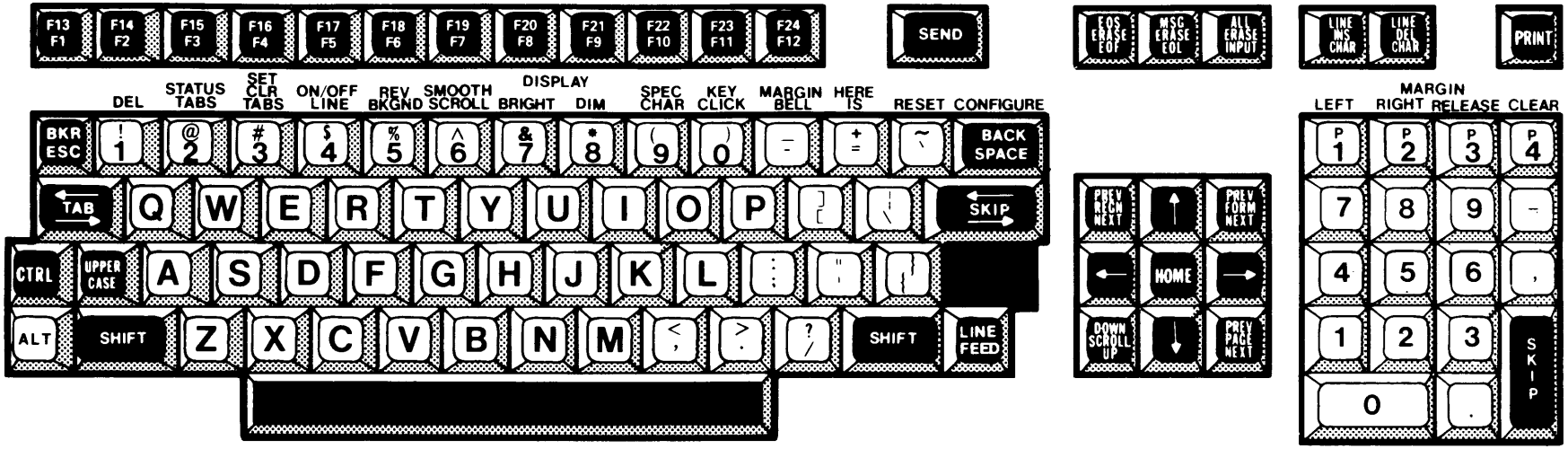
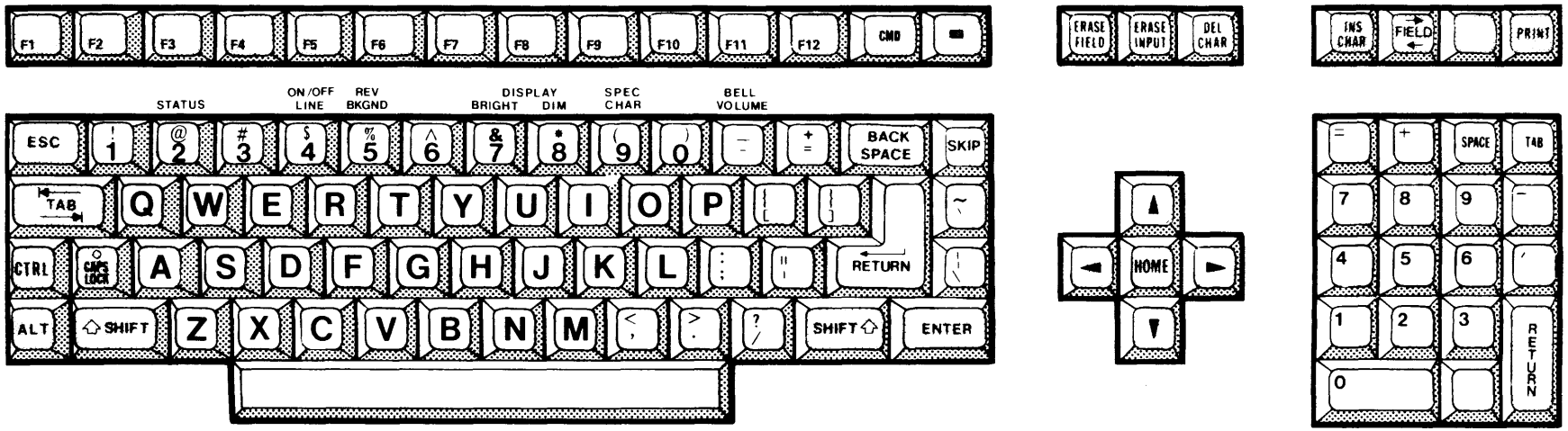


Figure A-2. 915 VDT Standard Keyboard Layout



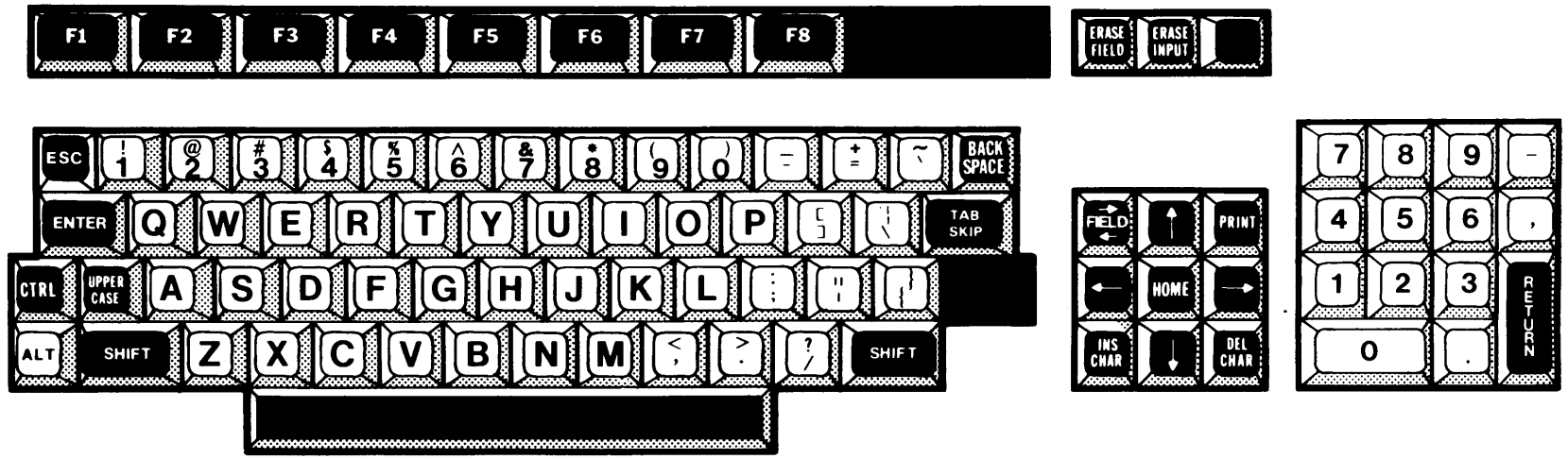
2284734 (11/14)

Figure A-3. 940 EVT Standard Keyboard Layout



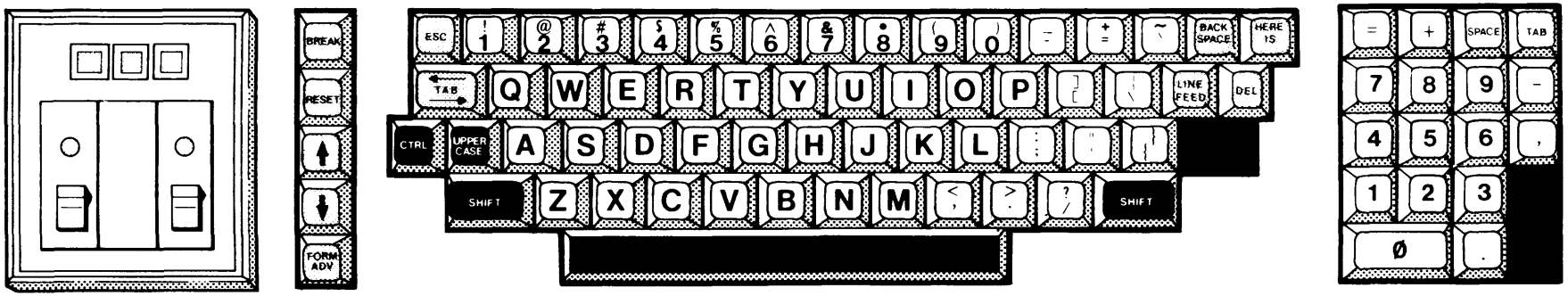
2284734 (12/14)

Figure A-4. 931 VDT Standard Keyboard Layout



2284734 (13/14)

Figure A-5. Business System Terminal Standard Keyboard Layout



2284734 (14/14)

Figure A-6. 820 KSR Standard Keyboard Layout

Appendix B

TIFORM Status Codes

TIFORM returns to the application two status codes indicating the success or failure of the last command executed. They are returned in the status block declared by the application on the last Declare Status Block call executed. The operating system (OS) status contains the code returned by the operating system on the last supervisor call (SVC); it is zero unless the last SVC failed, in which case it contains the reason for the failure. The form status contains TIFORM's internal status code. Form status values of nine (09) or less indicate that the operation was completed successfully, and are informative only. Values of ten (10) or greater indicate that a fatal error occurred. In the event of a fatal error, the form is automatically closed, and the Form Executor terminates.

Table B-1 gives the possible values in this field and their meanings.

Note that the TIFORM error codes are decimal and the OS error codes are hexadecimal. A right angle bracket (>) preceding a value indicates a hexadecimal value.

Table B-1. TIFORM Status Codes

Description	Code
Operation completed successfully:	
Complete read;terminated by exiting last field forward	00
Partial read; terminated by exiting first field backward	01
Partial read; terminated by an armed nonabort event key	02
Partial read; terminated by an armed abort event key	03
Error while opening terminal (SVC >00 OS status)	04
Open Form command issued while form is open	05
Partial read: terminated by TERMINATE READ [IMMEDIATELY]	06
Terminal disconnected	07
Fatal errors:	
Change ITC/IPC Communication command issued to connect to a new executor while application is currently connected to an executor	10
Command (not Open Form) issued while form is closed	12
Insufficient number of arguments in call argument list	13
Open Form parameter (file name/term-name) is too long	14
Segment too complex for terminal-dependent buffer size	15
Terminal read or write error (SVC > 00 OS status)	19

Table B-1. TIFORM Status Codes (Continued)

Description	Code
ITC/IPC error while sending to terminal	21
Executor has prematurely terminated (SVC > 0E OS status)	22
Failure to assign LUNO to ITC/IPC channel	23
Executor has prematurely terminated (SVC > 0E OS status)	24
Failure to close LUNO to ITC/IPC channel	25
Cannot start Executor (SVC > 2B OS status)	27
Failure to open LUNO to ITC/IPC channel	28
Failure to release LUNO to ITC/IPC channel	29
Invalid index or field count (nonnumeric, too big)	32
Invalid group, field, or variable name	33
Invalid segment name	34
Internal error in Form Executor	36
Excessive nesting of group definitions (10 levels max)	37
Invalid application command code	51
Application command code not currently supported	52
Form or segment not in specified file (SVC > 31 OS status)	53
Cannot load form/segment from file (SVC > 14 OS status)	54
First command not Open Form	55
Cannot assign LUNO to terminal (SVC > 00 OS status)	57
Terminal type specified in an Open/Change Form command not supported by TIFORM	58
Cannot assign LUNO to form file (SVC > 00 OS status)	59
Improperly formatted Arm Event Key command	60
Unable to Close/Release the terminal (SVC > 00 OS status)	61
Unable to execute the Print Key task (SVC > 2B OS status)	62
Invalid mode code specified	63
Unable to release the LUNO on the form program file during a Change/Close Form command	64
Application task terminated	98
Executor unable to send a message to the application due to ITC/IPC buffer congestion	99

Several form status codes are accompanied by an OS status code. This OS status code is the error status code from the second byte of an SVC block. For proper interpretation, an SVC opcode must be prefixed to this OS status code to yield an OS error number. The SVC opcode to append is shown in the table with each error that returns an OS status code. To interpret the resultant OS error number, refer to either the *DX10 Operating System Error Reporting and Recovery Manual* or the *DNOS Messages and Codes Reference Manual*, depending on the operating system you are using.

For example, if a form status code of 59 is returned with an OS status code of >27, then the OS error >0027 should be looked up in the appropriate OS error manual, yielding the explanation NO FILE DEFINED BY NAME SPECIFIED. Similarly, a form status code of 53 with an OS status code of >58 yields a OS error number of >3158 and the explanation NAME NOT IN DIRECTORY.

Frequently an application program combines the two codes into a single four-byte code for brevity. The first two bytes of such a combined code are the TIFORM status code. The last two bytes are the operating system status code.

Appendix C

TIFORM Error Codes

C.1 INTRODUCTION

This appendix contains the error codes and diagnostics that may be generated during the development and use of a form. Paragraph C.2 describes the TIFORM error messages, paragraph C.3 describes the FDLC diagnostics, and paragraph C.4 describes the ISGE error codes.

C.2 TIFORM ERROR MESSAGES

Messages provide information about the TIFORM operation in progress. TIFORM generates two types of messages as follows:

- Information Messages — Provide statistics or indicate the present phase of the TIFORM operation.
- Error Messages — Indicate an unrecoverable error was encountered and TIFORM terminates.

If the error is a file input or output error, the operating system displays the appropriate message and terminates the execution attempt.

All messages are preceded by a code indicating the source, type, and number of the message, as in the following:

aaa TIFORM nnnn < message>

The aaa is the message source, which can be one, two, or three characters long, as follows:

Source Code	Meaning
I	Informative message
U	User fatal error
S	System fatal error
H	Hardware fatal error
US	User or system fatal error
UH	User or hardware fatal error
SH	System or hardware fatal error
USH	User, system, or hardware fatal error

The nnnn is the message number.

Table C-1 lists the message numbers and their corresponding internal message codes.

Table C-1. TIFORM Error Messages

U	TIFORM-0012	COMMAND (NOT OPEN FORM) ISSUED WHILE FORM IS CLOSED. Explanation: The application issued a command other than an Open Form command while the form was closed. User Action: Check the application at the command that received the error for a previous Close command or for failure to issue an Open Form command.
U	TIFORM-0013	INSUFFICIENT NUMBER OF ARGUMENTS IN CALL ARGUMENT LIST. Explanation: The application issued a command that did not specify enough parameters. User Action: Check the application at the command that received the error against the definition of the command.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0014	OPEN FORM PARAMETER (FILENAME/TERM-NAME) IS TOO LONG. Explanation: The application issued an Open Form or Change Form command that specified a file name or a terminal name that contains too many characters. User Action: Check the application command that received the error for the length of the parameter used to specify the file name or the terminal name. The maximum length is 48 characters.
U	TIFORM-0015	SEGMENT TOO COMPLEX FOR TERMINAL-DEPENDENT BUFFER SIZE. Explanation: The segment name specified by the Prepare Segment command is so complex that the object code does not fit in the terminal-dependent buffer. User Action: A solution may be to compile the segment with the NOSYMT option.
U	TIFORM-0019	TERMINAL READ OR WRITE ERROR. Explanation: An error occurred when the TIFORM Executor tried to read from or write to the terminal or file. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0021	ITC/IPC ERROR WHILE TRYING TO SEND TO TERMINAL. Explanation: The TIFORM interface routines received an ITC/IPC error trying to write to the TIFORM Executor. User Action: Check the system log for errors related to this task.
U	TIFORM-0022	EXECUTOR HAS PREMATURELY TERMINATED. Explanation: An error occurred when the TIFORM Executor tried to read from or write to the terminal or file. User Action: Check Appendix B for further information concerning TIFORM status codes.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0023	FAILURE TO ASSIGN LUNO TO ITC/IPC CHANNEL. Explanation: An error occurred when the TIFORM Executor tried to assign a LUNO to the pathname of the ITC/IPC channel for the form device. User Action: Check the TIFORM directory .S$TIFORM for the existence or correctness of the pathname of the device ITC/IPC channel. You can use the SCI procedure RESETCHN on the TIFORM directory to reestablish the correct pathname.
U	TIFORM-0024	EXECUTOR HAS PREMATURELY TERMINATED. Explanation: An error occurred when the TIFORM Executor tried to read from or write to the terminal or file. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0025	FAILURE TO CLOSE LUNO TO ITC/IPC CHANNEL. Explanation: An error occurred when the TIFORM Executor tried to close the LUNO to the current ITC/IPC channel. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0027	CANNOT START EXECUTOR. Explanation: An error occurred while trying to start the TIFORM Executor. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0028	FAILURE TO OPEN LUNO TO ITC/IPC CHANNEL. Explanation: An error occurred when the TIFORM Executor tried to open the LUNO to the current ITC/IPC channel. User Action: Check Appendix B for further information concerning TIFORM status codes.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0029	FAILURE TO RELEASE LUNO TO ITC/IPC CHANNEL. Explanation: An error occurred when the TIFORM Executor tried to release the LUNO to the current ITC/IPC channel. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0030	UNABLE TO ASSIGN LUNO TO TIFORM PROG FILE. Explanation: The form program file specified on the Open Form cannot be accessed. User Action: Check the form program file parameter of the Open Form command that received the error for correctness.
U	TIFORM-0032	INVALID INDEX OR FIELD COUNT (NONNUMERIC, TOO BIG). Explanation: The index, field count, or cursor position specified on the indexed operation that failed was not a numeric value or was too large. User Action: Check the index, count, or cursor value in the application command that received the error.
U	TIFORM-0033	INVALID GROUP, FIELD, OR VARIABLE NAME. Explanation: The application command specified a segment name, group name, field name, or variable name not available with the currently prepared segment. User Action: Check the application command that received the error to ensure that the correct segment name was prepared. Check the application to ensure that the correct segment, group, field, or variable name was referenced by the operation in error.
U	TIFORM-0034	INVALID SEGMENT NAME. Explanation: The Prepare Segment command issued by the application specified a segment name not available with the currently opened form. User Action: Check the application command that received the error to ensure that the correct segment name was requested.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0036	INTERNAL ERROR IN FORM EXECUTOR. Explanation: The TIFORM Executor is in an abnormal error condition. User Action: Attempt to reproduce the problem, then notify the TI customer representative of this condition.
U	TIFORM-0037	EXCESSIVE NESTING OF GROUP DEFINITIONS (10 LEVELS, MAX). Explanation: The group name specified on the previous command is defined as containing groups, which also are defined as containing groups. This nesting exceeds ten levels. User Action: Redefine the groups involved in the appropriate segment.
U	TIFORM-0051	INVALID APPLICATION COMMAND CODE. Explanation: An invalid application command code was specified in a TIFORM command. User Action: The specified call is not a legal TIFORM command. Correct the application involved. The link map of the application may have unresolved references related to this illegal command.
U	TIFORM-0052	APPLICATION COMMAND CODE NOT CURRENTLY SUPPORTED. Explanation: An application command code that is not currently supported was specified in a TIFORM command. User Action: The specified call is not a valid TIFORM command. Correct the application involved. The link map of the application may have unresolved references related to this illegal command.
U	TIFORM-0053	FORM OR SEGMENT NOT IN SPECIFIED FILE. Explanation: The application issued either an Open Form or Prepare Segment command containing a form name or segment name, respectively, that is not contained in the form program file specified in the Open Form command. User Action: Check the application command to ensure that the correct and valid program name was specified on the Open Form command. Also, check the Open Form or Prepare Segment command that received the error to ensure that the correct form name or segment name, respectively, was specified.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0054	CANNOT LOAD FORM OR SEGMENT FROM FILE. Explanation: An error occurred when trying to access the form or segment named in the Open Form or Prepare Segment command, respectively. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0055	FIRST COMMAND NOT OPEN FORM. Explanation: The TIFORM Executor received a command other than Open Form while the form was closed. User Action: Check the application, at the command that received the error, for a previous Close command or for failure to issue an Open Form command.
U	TIFORM-0057	CANNOT ASSIGN LUNO TO TERMINAL. Explanation: The TIFORM Executor cannot gain access to the terminal specified in the Open Form command. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0058	TERMINAL TYPE SPECIFIED IN OPEN/CHANGE FORM COMMAND, NOT SUPPORTED. Explanation: The TIFORM Executor does not support the type of terminal specified in the Open Form or Change Form command. User Action: Attempt this operation on a terminal type referenced in this manual. If the manual specifies that TIFORM supports this terminal type, notify the TI customer representative.
U	TIFORM-0059	CANNOT ASSIGN LUNO TO FORM FILE. Explanation: A system error occurred when the TIFORM Executor tried to assign a LUNO to the form program file specified. User Action: Check Appendix B for further information concerning TIFORM status codes.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0060	IMPROPERLY FORMATTED ARM EVENT KEY COMMAND. Explanation: The application issued an Arm Event Key command that indicated a buffer that is not formatted correctly. User Action: Check the application command that received the error to verify that the buffer specified on the call contains an event key list formatted as specified in Section 5.
U	TIFORM-0061	UNABLE TO CLOSE OR RELEASE THE TERMINAL. Explanation: An error occurred during the attempt to close or release the terminal or file currently used as the forms display device. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0062	UNABLE TO EXECUTE THE PRINT KEY TASK. Explanation: The TIFORM Executor received an error trying to bid the Print Key task. User Action: Check Appendix B for further information concerning TIFORM status codes.
U	TIFORM-0063	INVALID MODE CODE SPECIFIED IN A CONTROL FUNCTIONS COMMAND. Explanation: The application issued a Control Functions command that indicated a buffer containing an invalid control code. User Action: Check the application command that received the error to verify that the buffer specified on the call contains a control code formatted as specified in Section 5.
U	TIFORM-0064	UNABLE TO RELEASE LUNO OF FORM PROGRAM FILE DURING CHANGE/ CLOSE FORM. Explanation: The TIFORM Executor received an error trying to release the form program file during a Close or Change Form command. User Action: Check Appendix B for further information concerning TIFORM status codes.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0097	EXECUTOR HAS TERMINATED... Explanation: The TIFORM Executor abnormally terminated due to external circumstances. User Action: Check the system log for further information.
U	TIFORM-0098	APPLICATION TASK HAS TERMINATED. Explanation: The application task abnormally terminated due to circumstances unknown to the TIFORM Executor. User Action: Check the application recovery process or the system log for more information.
U	TIFORM-0099	EXECUTOR UNABLE TO SEND TO APPLICATION DUE TO ITC/IPC CONGESTION. Explanation: The ITC/IPC buffers are full, thus the TIFORM Executor is unable to send any messages to the application. User Action: Execute the CLEAR task from the program file .S$TIFORM.PROG using the Execute Task (XT) command.
U	TIFORM-0100	Shut 'er Down Clancey She's a Pumping Mud. Explanation: An error occurred in the TIFORM Executor that is not identifiable. User Action: Please call the TI customer representative.
U	TIFORM-0101	OUTPUT PROCESSING ERROR, VERIFY ALL DATA. Explanation: An error occurred during final validation processing. User Action: Check all data just received for correctness.
U	TIFORM-0102	FIELD I/O ABORT ERROR. RE-ENTER DATA. Explanation: The TIFORM Executor received an I/O abort event key sequence. User Action: Reenter the data in the appropriate field.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0110	SCREEN HAS BEEN SENT TO PRINTER ?1. Explanation: The Print Key task successfully sent the current screen image to the specified printer. User Action: No user action necessary.
U	TIFORM-0111	SCREEN HAS BEEN SENT TO A FILE ?1. Explanation: The Print Key task successfully sent the current screen image to the specified file. User Action: No user action necessary.
U	TIFORM-0112	CANNOT ACCESS PRINTER ?2. Explanation: The Print Key task cannot gain access to the specified print device. User Action: Check the device specified as the print device for this terminal to ascertain the problem.
U	TIFORM-0113	CANNOT ACCESS LOGICAL NAME: TIFRMPRT. Explanation: The Print Key task attempted to access the standard logical name TIFRMPRT but received an error. User Action: The logical name TIFRMPRT must be assigned to the spooler task if the specified print device is declared a spooler resource.
U	TIFORM-0114	CANNOT ACCESS SPOOLER. Explanation: The Print Key task received an error attempting to access the spooler via the logical name TIFRMPRT. User Action: Check spooler status.

Table C-1. TIFORM Error Messages (Continued)

U	TIFORM-0128	TIFORM DISCONNECT ON A SEND. Explanation: The TIFORM Executor received an error trying to send data to the application via the ITC/IPC channel. User Action: Check the application and system log for abnormal task termination. Also, ensure that the task is issuing a Close Form command before it terminates.
U	TIFORM-0129	TIFORM DISCONNECT ON A RECEIVE. Explanation: The TIFORM Executor received an error trying to receive data from the application via the ITC/IPC channel. User Action: Check the application for abnormal termination or termination without issuing a Close Form command. Also check the system log for task errors related to the application task.
U	TIFORM-0130	ERROR TRANSLATING STATION NUMBER FOR PRINT KEY TASK. Explanation: The Print Key task attempted to read the screen image of a device name that is too large or nonnumeric. User Action: Check the application for previous TIFORM status codes.

C.3 FDL COMPILER DIAGNOSTICS

During the one-pass compilation process, the FDLC writes diagnostics to two user-specified files. One of these files is the listing file where the diagnostics are included immediately following either the statement that contained the error or the statement that was being processed at the time an error condition was discovered. The other file is the error file that contains an image of each erroneous statement followed by the diagnostic message. If you do not specify an error file, the terminal local file is used as a default.

An error message is always preceded by eight asterisks. Following the asterisks and preceding the text of the error message, there are two numbers that represent the error message number and the severity level, respectively. These numbers are written in the format xxx,y where xxx represents the error number and y the severity level. Table C-2 reproduces the error number, severity level, and text of each error message and provides a short explanation of the error. The conventions used for severity levels are as follows:

Severity Level		Meaning
WARNING		
	1	Lowest warning
	2	
	3	
	4	Highest warning
ERROR		
	5	Current statement discarded
	6	Current statement discarded
	7	Current statement discarded
	8	Current statement/block structure discarded
	9	Immediate abort

The severity levels are intended for future enhancements to the error recovery strategies of the FDLC. At this time they take on the following significance. All warning levels are equivalent and indicate that despite something being amiss, the compilation process remains unaffected. Severity levels five through eight are essentially indistinguishable insofar as the action the compiler takes, but their numerical rating is hopefully indicative of the seriousness of the error. Severity level nine, however, is special and does in fact produce an immediate abort.

When an error in a statement requires that the statement be discarded, the compiler does so by scanning for the first occurrence of the sequence of end-of-statement characters (that is, a period followed by a blank). Compilation begins again with the next record containing an FDL statement. If the end-of-statement characters were inadvertently omitted from the statement being discarded, the subsequent statement is lost during the resynchronization process. The loss of this additional statement, however, is not apparent from the error message file and it may, in fact, produce misleading diagnostics. In general, omission of the end-of-statement characters may cause peculiarities in compilation results and you are cautioned to be alert to this situation. When attempting to decipher peculiar diagnostics, first ascertain whether any FDL statements are missing the sequence of end-of-statement characters.

The FDLC termination message displayed at the user's terminal can be one of two formats, depending upon the success of attempts to acquire user files. If a file error occurs during initialization, an immediate abort occurs and the termination message is one of error messages numbers 1, 2, 6, or 7 in Table C-2. If the initialization process is successful, the FDLC termination message is in the following standard format:

FDL ERROR COUNTS: xxx xxx xxx xxx/xxx xxx xxx xxx xxx

Each xxx represents the error count for a particular severity level. The severity levels are represented from left to right in increasing order from one to nine with the slash separating warnings from errors. Thus, the following termination message indicates that three warnings have been issued (two of severity level two and one of severity level four) and nine errors have been detected (five of severity level six and four of severity level seven).

FDL ERROR COUNTS: 000 002 000 001/000 005 004 000 000

Table C-2 contains the text, error number, and severity level of all FDLC diagnostics. Ampersands indicate variable length compiler-supplied parameters. They represent such items as line numbers, column numbers, file names, operating system or SCI error numbers, and user-specified names, keywords, numerics, and so on. It should be apparent in most cases which parameter item the ampersand represents.

Table C-2. FDL Compiler Diagnostics

Error Number/Severity Level/Message Text	Explanation
001,9.....OS error & assigning a LUNO to & 002,9.....OS error & opening & 003,9.....OS error & reading & 004,9.....OS error & writing & 005,9.....OS error & closing &	These five messages all report OS SVC errors generated by trying to perform the indicated action against the specified file. See the error manual of the For an explanation of the error code, refer to either the *DX10 Operating System Error Reporting and Recovery Manual (Volume VI)* or the *DNOS Messages and Codes Reference Manual*, depending on the OS you are using.
006,9.....SCI error & opening & as msg file 007,9.....SCI error & getting parameter number &	These two messages report an SCI error while trying to perform the indicated action against the specified file. For an explanation of the error code, refer to either the *DX10 Operating System Error Reporting and Recovery Manual (Volume VI)* or the *DNOS Messages and Codes Reference Manual*, depending on the OS you are using.
008,7.....Column &: expected FDL verb; found —&—	The first token of an FDL command must be one of a special set of keywords called verbs. If the first token is not a verb, this message reports the column in which the first token starts and the erroneous token.
009,7.....Column &: expected end-of-statement, found —&—	An FDL statement must end with an end-of-statement token, a period followed by a blank. If the FDLC finds some other token where it expects to find an end-of-statement, this message reports the column in which the token starts and the erroneous token. This message is most commonly caused by forgetting to end an FDL statement with a period. In this case, the first token of the next statement is reported as the erroneous token. The FDLC's error recovery then discards the rest of the next statement looking for an end-of-statement token.

Table C-2. FDL Compiler Diagnostics (Continued)

Error Number/Severity Level/Message Text	Explanation
010,7.....Column &: —&— ill-formed token	The rules defining the legal tokens of the FDL language are quite rigid; for example, numbers cannot contain alphabetic characters, alphanumeric words cannot contain double quotes, and so on. This message reports that while trying to isolate the next token of an FDL statement, the FDLC found a character that violates FDL's rules. The message reports the column where the ill-formed token starts and the ill-formed token. The rightmost character of the ill-formed token is the character that made it ill-formed.
011,4.....This attribute specified previously for this field	Each attribute can be specified only once for a given field. This message reports that the attribute specified by the current statement has already been specified at least once for this field.
012,6.....Part number —&— must be 6 digits 013,6.....Revision number —&— must be 2 digits	The part number and revision number on a FORM statement must be precisely six and two digits long, respectively.
014,6.....Bad row number —&—	A row number must be an unsigned decimal integer between 1 and 24 inclusive.
015,6.....Name —&— too long	A name in FDL must be no more than six characters long.
016,6.....Bad column number —&—	A column number must be an unsigned decimal integer between 1 and 80 inclusive.
017,8.....More than 255 "M" statements in this mask	No more than 255 background text statements are allowed within a single segment mask or field mask block. This message appears following the 256th background text statement.
018,6.....Ill-formed function key number —&— 019,8.....—&— contains too many digits for a function key number.	Function key numbers in an FKEYS statement must be two digit unsigned decimal numbers.
020,7.....OS error & trying to install & as an overlay	At the end of each segment, segment mask, and form, the FDLC installs an overlay with the name of the form/segment/mask into the program file specified in the XFDLC command. This message reports the OS SVC error and the name of the overlay when such an installation fails.

Table C-2. FDL Compiler Diagnostics (Continued)

Error Number/Severity Level/Message Text	Explanation
021,6.....Improper decimal digit in —&—	Self-explanatory.
022,2.....Incompatible THEN/ELSE clauses	There are two kinds of IF CONDITION statements, that is, conditional branching and conditional attribute selection. They differ only in the objects of their THEN and ELSE clauses. Conditional branching specifies THEN/ELSE GOTO <name>; conditional attribute selection specifies THEN/ELSE EDITS = <name>. This message reports a mixture of the two kinds of THEN and ELSE clauses. This message also appears if neither the THEN nor the ELSE clause is specified.
023,6.....Reference to —&— incompatible with ref on line & 024,6.....Reference to —&— incompatible with def on line & 025,6.....Definition of —&— incompatible with ref on line & 026,6.....Double definition: —&— already defined on line &	These messages report inconsistencies in the definition and/or use of a name. Since the FDLC is a one-pass compiler, the "ref" line number reported is that of the last reference to the specified name.
027,6.....Illegal statement in current context: &	Appendix F lists the contexts of FDL and which statements are legal in which contexts. This message reports that the current statement is not legal in the current context and states what the current context is.
028,6.....—&— not defined previously as an array	A A <name>(<row>,<col>) reference in a group specification requires that <name> already be defined as a field array. This message reports that this is not the case.
029,4.....Attributes overridden by an OUTPUT statement	An output field cannot have editing or processing attributes. This message reports that attributes were specified in conjunction with the OUTPUT statement and they were ignored.
030,6.....—&— referenced on line & but never defined	When the end of a segment block occurs, the FDLC scans its name table for undefined names. This message is used to report each undefined name. The line number given is that of the last reference to the specified name.
032,7.....END name —&— does not match name defined on line &	The name on each END statement must match the name defined on the corresponding beginning block statement. This message reports a mismatch, stating the name on the END statement and the line number of the beginning block statement.

Table C-2. FDL Compiler Diagnostics (Continued)

Error Number/Severity Level/Message Text	Explanation
033,4.....Segment longer than & bytes	Self-explanatory.
034,7.....Column &: expected &; found —&— 035,7.....Column &: expected "&"; found —&—	These messages report a syntax error. The FDLC was looking for a particular keyword or syntactic construct but found something else. These messages report the keyword/construct the FDLC expected and the keyword/construct the FDLC found. Where there are several possible legal syntactic constructs, these messages only report the last one as the expected construct; the construct actually found was none of the legal possibilities.
036,7.....Column &: null literals are not allowed 037,7.....DECIMAL attribute is illegal on LEFT justify statement	Self-explanatory.
038,7.....—&— contains too many characters for a CHAR LIST item	A CHARACTER LIST statement consists of single character items (possibly quoted) separated by commas and/or ellipsis. This message reports that the specified item is more than one character long.
039,7.....—&— does not match preceding SEGMENT MASK —&—	The segment mask name defined in the segment statement does not match the name defined in the segment mask statement.
040,1.....Control mode settings updated in form root &	If the control modes are updated/edited in the form root, a warning message is displayed notifying the change.
041,8.....END processing omitted due to severe error on line &	Certain errors within a FIELD block are severe enough to force the FDLC to suppress the processing normally performed on an END FIELD statement. This message is actually a warning that an earlier error invalidated the compiled object form.
042,6.....Field length specified incorrectly or not specified at all	A field length must be an unsigned nonzero positive decimal integer. Every POSITION statement must specify a length.
043,8.....Array increments out of bounds	A field array must not specify a position outside the bounds of the screen. The size of the screen is determined from the specified DEVICE type.
044,9.....Name table overflow	Too many names and/or field array elements were specified. Approximately 400 names/array elements can be defined but the precise number depends on the mix of named items and field array elements.

Table C-2. FDL Compiler Diagnostics (Continued)

Error Number/Severity Level/Message Text	Explanation
045,6.....DISPLAY attribute specification incorrect	A field cannot have both true graphics (GRAPHICS INPUT) and virtual graphics (DISPLAY GR = Y)
046,8.....& (line &) is SAME AS & (line &), an OUTPUT field	A SAME AS statement references a field that has the OUTPUT attribute. This is invalid.
047,6.....Mask/segment uses rows/columns outside device limits (&/&)	A row/column number exceeds the device's limits.
048,8.....The following fields/edit sets are involved in a SAME AS loop...	By using the SAME AS attribute, a loop has been created.
049,1.....—&— defined on line &	The symbol is caught in a SAME AS loop.
050,4.....—&— is an invalid device type	TIFORM does not currently support the named device.
051,6.....This field extends beyond column &	The length of the field causes one or more column positions to appear past the right boundary of the screen. The length of the lines of the screen is determined from the specified DEVICE type.
052,6.....This mask element extends beyond column &	The length of the element causes one or more characters to appear past the right boundary of the screen. The length of the lines of the screen is determined from the specified DEVICE type.
053,6.....Field name —&— is & chars too long for its array	The name of a field that has the ARRAY attribute along with the array element suffix appended by the FDLC exceeds six characters. You must shorten the field name by the specified number of characters.
054,7.....—&— has row/column index out of range	A group member that is an array reference has an index that references a nonexistent element of the array.
055,9.....A & named & already exists	All names of segments/segment mask/forms must be unique.
056,7.....Segment mask not specified with segment	A segment mask is not specified in the SEGMENT statement.
057,8.....Cannot compile segment mask & without segment	You cannot compile a segment mask by itself. You must compile a segment with it.
058,6.....Cannot compile negative attribute with SAME AS	If you use the SAME AS attribute in a field/edit set, you cannot use negative attributes in conjunction with it.

Table C-2. FDL Compiler Diagnostics (Continued)

Error Number/Severity Level/Message Text	Explanation
059,7.....Error in attempting to delete overlays	When the DELSEG option is in effect, the compiler either cannot find the appropriate overlay or the overlay is not a segment or segment mask.
060,1.....Form root has been expanded to —&— bytes.	For optimum execution, the form root should be no greater than 2000 bytes.
62,6.....A field is positioned outside of device limits.	A field cannot be positioned outside of the row and column limits of the device in use.
63,8.....Segment compilation errors, overlays not installed.	When fatal errors occur, the segment mask overlay and the segment overlay are not installed, and the form root is not modified.
65,6.....ON COMPLETION destination must be current field.	ON COMPLETION processing for a COPY TO statement can take place on the the current field (*) only.

C.4 ISGE ERROR MESSAGES

This appendix presents all the error messages that the ISGE produces. The text of each message is shown with an explanation of the error and suggestions for recovery.

Each ISGE error message requires confirmation. A blinking cursor is displayed to the right of the message. You must press the Return key or the Enter key before the ISGE continues.

The value nnnn in a message represents an error code. The leftmost two digits contain the ISGE error code; the rightmost two digits contain the SVC error code (if any). The ISGE error codes are discussed immediately following the explanation of the message in which they may appear. Depending on the OS you are using, you can find the definition of SVC error codes in either the *DX10 Operating System Error Reporting and Recovery Manual (Volume VI)* or the *DNOS Messages and Codes Reference Manual*.

C.4.1 Intermediate Segment File Recovery

Various fatal errors could cause the ISGE to abort before the ISGE session can be terminated cleanly. Depending on the error, special action may be necessary to recover the intermediate segment file, even though saving the intermediate segment file was specified. If an abort nullifies an intermediate segment file save, there are two places where a copy of the aborted session's intermediate segment file might be found. First, the ISGE keeps its working copy of the intermediate segment file for terminal STnn in the following file:

S$TIFORM.ISGE.STnn

This file is deleted if the ISGE terminates normally, but after an abort it may still exist. If no other ISGE session has been started on terminal STnn subsequent to the aborted session, this file contains the aborted session's intermediate segment file.

The other place to look for the intermediate segment file is in a file of the following name:

S$TIFORM.ISGE.aaa

The value aaa is a unique three-character name between AAA and ZZZ. If no intermediate segment file save was specified, but FDL construction was specified, the ISGE creates a new "aaa" name and copies the session's intermediate segment file into that file before starting the FDL construction task in background. A successful FDL construction deletes this file before terminating. If the FDL construction task aborts, the session's intermediate segment file is left in this file. To recover the aborted session's intermediate segment file, you should first execute the List Directory (LD) command on .S$TIFORM.ISGE to get the names of all such files. You must determine which file is the applicable intermediate segment file experimentally.

Once you locate the aborted session's intermediate segment file, you should copy it into some other file of your choice. The ISGE reuses its files whenever it runs so you must save them explicitly if you want to retain them.

C.4.2 List of Messages

CURSOR IS NOT POSITIONED IN AN ACTUAL FIELD

While in screen drawing mode, you made an attempt to delete or move a field but the cursor was not positioned on a field when you pressed the function key. Either reposition the cursor so it lies in a field before you attempt the operation again, or supply row/column coordinates other than the cursor position.

EDITOR ABORTING

The ISGE is aborting the session because of a fatal error from either the TIFORM Executor or the file handler portion of the ISGE. This message is displayed after several of the following fatal error messages.

<nnnn> FATAL ERROR WHILE DECOMPILING SEGMENT

The ISGE encountered a fatal error situation while decompiling a segment and its mask. The ISGE terminates after reporting the error.

If the error is the result of an invalid program file pathname, the following error code is returned:

Code	Description
62xx	LUNO assignment error

If an invalid segment name was entered, one of the following error codes is returned:

Code	Description
61xx	Load overlay error
63xx	Overlay number error

The following error codes indicate that the segment's object form is bad:

Code	Description
4100	Cannot decompile 3.1.0 segments
4200	Overlay loaded is not a segment mask
4300	Matching relative address not found in name table
4400	Length list definition incorrect
4500	Segment does not contain a valid symbol table

If the segment to be decompiled exceeds any of the ISGE's size limitations, one of the following error codes is returned:

Code	Description
5100	Some item is longer than 578 bytes.
5200	Segment has more than 320 different names.
5300	Edit sets are nested more than 10 levels deep.
5400	Segment has more than 100 fields.
5500	Object for segment or mask is more than 4000 bytes long.
5600	The number of named user-specified error messages exceeds 100.

FATAL FILE HANDLER ERROR. CMD BLOCK FOLLOWS.

This message indicates an internal error within the logic of the ISGE. The ISGE aborts the session after displaying this message. After this message is displayed, the ISGE's internal command block is displayed. Digits 7 through 10 of the command block contain one of the following error codes:

Code	Description
9100	Invalid command sent to Decompiler
9200	Invalid command received from ISGE
9300	Invalid command received from Decompiler
94xx	I/O operation on ISGEFILE failed
94xx	I/O operation on PATHFILE failed

FDL BUILDER CANNOT OPEN FDL SOURCE FILE. OS ERROR < nn> ON < pathname> .

You specified an invalid pathname for the FDL source file. The value < nn> is a COBOL I/O error code described in the appropriate COBOL reference manual as listed in the Preface. You can retrieve the intermediate segment file containing the segment just generated or edited from one of the following three places:

- If you specified an intermediate segment file pathname at the termination of the ISGE, the intermediate segment is in that file.
- If you terminated the ISGE with a COMPILE FDL command and you did not specify an intermediate segment file pathname, the intermediate segment is in the system file .S$TIFORM.ISGE.STxx, where xx is the ID of the terminal on which the ISGE is running.
- If you terminated the ISGE with a BUILD FDL command and you did not specify an intermediate segment file pathname, the intermediate segment is in a system file under the directory .S$TIFORM.ISGE. You must execute the List Directory (LD) command on this directory in order to determine the file's specific pathname. The file name to look for in the list directory is a three character name in the range AAA – ZZZ. If more than one of the file names falls in this category, the one containing the intermediate segment for which the error message was generated is most likely to be the file name having the highest value. To verify that you selected the correct intermediate segment file, you can perform the Show File (SF) command on the file. The segment name should be in the first record of the file.

Once you determine the intermediate segment file's pathname, you should copy the intermediate segment to one of your files. You can execute the ISGE again, retrieve the intermediate segment, and specify a valid FDL source file at termination.

FIELD OVERLAPS AN EXISTING FIELD

You made an attempt to insert a field on top of an existing field. You can abort the insert field command with the Command key, or you must supply alternate row/column coordinates.

12xx FILE ERROR WHILE OPENING INTERMEDIATE FILE

You specified an invalid pathname for an intermediate segment file, and the ISGE is unable to open the file. The ISGE prompts for another pathname, or the session may be aborted.

< nnnn> FILE ERROR WHILE TERMINATING

You specified an invalid pathname for the file in which the intermediate segment is to be saved at the end of the session. The ISGE prompts for another pathname. One of the following error codes is returned with this message:

Code	Description
11xx	Error in rename executed to save the intermediate segment file
13xx	Invalid pathname specified to save the intermediate segment file

FORM ERROR. STATUS BLOCK FOLLOWS.

This message indicates an internal error in the logic of the ISGE. The ISGE received an error from the TIFORM Executor. The 40-byte TIFORM status block is displayed after this message appears. The fatal error code is in the first four characters of the status block. The ISGE aborts the session after displaying the error message.

INVALID ROW, COLUMN, OR LENGTH VALUE

This message indicates that you specified a row, column, or length value that exceeds the dimensions of the VDT screen.

ISGE WAS UNABLE TO DELETE TEMPORARY FILE: < pathname> SVC ERROR CODE = xx.

The ISGE created a file for the FDL compiler to use, and the FDL compiler was unable to delete it before terminating. This error does not affect the results of the ISGE in any way. If this message is displayed, you should perform a delete file operation on the file whose pathname is specified to avoid cluttering the system with unnecessary files.

ROLL MUST BE "U" OR "D"

On either an insert or delete lines command, you specified a roll value other than U or D. You can abort the command with the Command key, or you must specify the roll value as U or D.

Appendix D

Examples of FDL Form Definitions

This appendix contains several examples of segments written in the Form Definition Language (FDL). They illustrate many of the possible FDL constructs.

These examples reside in the directory .S$TIFORM.TESTFORM on the installation disk. You can compile them by using the FDL Compiler and then execute them by using the Form Tester.

Note that all comments are legally placed throughout the segments.

The first segment is the FDL produced during the ISGE tutorial in Section 4. The source that the ISGE produces should be exactly like this segment, without the comments.

EXAMPLE

```
SEGMENT MASK ORDERM,CLEAR=Y.  (Define the screen mask ORDERM, clear
.                              the screen before displaying.)
    DISPLAY GR =Y.            (Turn graphics mode on to draw
    M(003,005)                 the border.)

'BLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL'
 .
    M(003,075) 'C'.
    M(004,005) 'I'.
    M(004,075) 'I'.
    M(005,005) 'I   '.
    DISPLAY GR =N.                     (Turn the graphics mode of write
    M(005,009) 'FRED''S RACQUET SHOP'.  requested text on screen.)
    M(005,058) 'INVOICE #'.
    DISPLAY GR =Y.             (Graphics mode on.)
    M(005,075) 'I'.
    M(006,005) 'I'.
    M(006,075) 'I'.
    M(007,005) 'I   '.
    DISPLAY GR =N.             (Graphics mode off.)
    M(007,009) 'PART #'.
    M(007,022) 'DESCRIPTION'.
    M(007,037) 'QTY'.
    M(007,048) 'PRICE'.
    M(007,058) 'TOTAL'.
    DISPLAY GR =Y.             (Graphics mode on.)
    M(007,075) 'I'.
    M(008,005) 'I'.
    M(008,075) 'I'.
    M(009,005) 'I'.
    M(009,075) 'I'.
    M(010,005) 'I'.
```

```
    M(010,075) 'I'.
    M(011,005) 'I'.
    M(011,075) 'I'.
    M(012,005) 'I'.
    M(012,075) 'I'.
    M(013,005) 'I'.
    M(013,075) 'I'.
    M(014,005) 'I'.
    M(014,075) 'I'.
    M(015,005) 'I'.
    M(015,075) 'I'.
    M(016,005) 'I'.
    M(016,075) 'I'.
    M(017,005) 'I'.
    M(017,075) 'I'.
    M(018,005) 'I   '.
    DISPLAY GR =N.            (Graphics mode off.)
    M(018,009) 'EMPLOYEE NAME'.
    DISPLAY GR =Y.            (Graphics mode on.)
    M(018,075) 'I'.
    M(019,005) 'I'.
    M(019,075) 'I'.
    M(020,005)
 'DLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL'
 .
    M(020,075) 'E'.
.
END SEGMENT MASK ORDERM.      (End screen mask drawing.)
.
.
.

SEGMENT ORDERS,(ORDERF),ORDERM. (Define segment ORDERS, put into form root
.                                    ORDERF, and use segment mask ORDERM.)
.
.
FIELD NVOICE.                   (Field named NVOICE.)
    POSITION (5,67)L4.  (Positioned at line 5, column 67, 4 bytes long.)
    REQ.                (A value must be entered into this field.)
    NUMERIC.            (The value must be numeric.)
END FIELD NVOICE.       (End of field NVOICE.)
.
FIELD PARTN.            (Field named PARTN.)
    POSITION (8,9)L6.   (Line 8, column 9, 6 bytes long.)
    DISPLAY BR=Y.       (Highlight this field's value.)
    AUTOSKIP.           (Automatically move cursor to next field.)
    CHAR LIST=DIGIT.    (Value must be between 0 and 9.)
    LENGTH LIST=PNLEN;DIAGNOSTIC=ERRPN.
                        (Value must be 4 digits in length, if not
.                        a user error message will appear.)
END FIELD PARTN.
.
```

```
FIELD DESCRP.                       (Field named DESCRP.)
    POSITION (8,22)L11.             (Line 8, column 22, 11 bytes long.)
    DISPLAY MASK DSCRIP,POSTCLEAR. (Show field mask DSCRIP on screen.)
    TABLE LIST=THINGS.              (Value must be specified in this list.)
END FIELD DESCRP.
.
FIELD QNTITY.                       (Field named QNTITY.)
    POSITION (8,37)L3.              (Line 8, column 37, 3 bytes long.)
    DISPLAY BR=Y.                   (Highlight this field.)
    RANGE LIST=RGEQTY.              (Value must be between 1 and 99999.)
END FIELD QNTITY.
.
FIELD PRICE.                        (Field named PRICE.)
    POSITION (8,48)L6.              (Line 8, column 48, 6 bytes long.)
    DISPLAY BR=Y.
    SIGNED NUMERIC, FILL=' ', DEC=2.
                                   (Value will be signed, justified right,
.                                   filled with leading blanks, and will
.                                   have two digits to the right of the
.                                   decimal point.)
    COPY TO TOTAL.       (Copy the value to the field TOTAL
    CHAR LIST=MONEY.      0 through 9, + and . are the only valid chars.)
END FIELD PRICE.
.
EXTERNAL TOTAL.          ( Allow field TOTAL to be available to the
.                          application.)
FIELD TOTAL.                    (Field named TOTAL.)
    POSITION (8,58)L7.          (Line 8, column 58, 7 bytes long.)
    DISPLAY BR=Y.
    RANGE LIST=RANGE$.      (This statement ignored because of conditional
.                           statement.)
    IF COND BONUS$ ON PRICE  (If the value is between 50 and 99999 in
        THEN EDITS=DOTHIS    field PRICE then select attributes from
        ELSE EDITS=DOTHAT.   DOTHIS;otherwise select attributes from
END FIELD TOTAL.              DOTHAT.)
.
FIELD YRNAME.                   (Field named YRNAME.)
    POSITION (18,23)L20.        (Line 18, column 23, 20 bytes long.)
    REQ.
    CHAR LIST=YOUWHO.           (A through Z are the only valid
END FIELD YRNAME.                characters.)
.
FIELD MASK DSCRIP,CLEAR=N.   (Field mask DSCRIP, do not clear screen before
    DISPLAY BR=Y.             showing:  highlight this mask.)
    M(9,12) 'DESCRIPTION ITEMS ARE:'. (Mask text that will be displayed.)
    M(10,12) 'HAT, RACKET, CLOTHING, BALLS, SHOES'.
END FIELD MASK DSCRIP.
.
FIELD MASK COMISS,CLEAR=N.      (Field mask COMISS.)
    DISPLAY BR=Y.
    M(10,44) 'CONGRATULATIONS! YOU WILL'.
    M(11,44) 'RECEIVE A 10% COMMISSION'.
END FIELD MASK COMISS.
```

```
.
FIELD MASK TOOBAD,CLEAR=N.        (Field mask TOOBAD.)
    M(14,33) 'BETTER LUCK NEXT TIME!'.
END FIELD MASK TOOBAD.
.
.
LIST CHAR DIGIT=0..9,BLANK.
.
LIST CHAR MONEY=0..9,'.','+',BLANK.
.
LIST CHAR YOUWHO=A..Z,BLANK.
.
LIST RANGE RGEQTY=IN,1/99999.
.
LIST RANGE RANGE$=IN,50/99999.
.
LIST LEN PNLEN=4.
.
LIST TABLE THINGS=IN,'HAT','RACKET', 'CLOTHING', 'BALLS',
                   'SHOES'.
.
CONDITION BONUS$.              (Condition named BONUS$.)
    RANGE LIST=RANGE$.
END CONDITION BONUS$.
.
EDIT SET DOTHIS.               (Edit set named DOTHIS.)
    DISPLAY MASK COMISS,POSTCLEAR.
.                             (If condition was true display field mask.)
END EDIT SET DOTHIS.
.
EDIT SET DOTHAT.               (Edit set named DOTHAT.)
    DISPLAY MASK TOOBAD,POSTCLEAR.
.                              (If condition was false, display field mask.)
END EDIT SET DOTHAT.
.
ERROR MESSAGE ERRPN='PART NUMBER MUST BE 4 DIGITS'.
.                              (User defined error message.)
.
END SEGMENT ORDERS.            (End of segment ORDERS.)
```

EXAMPLE

```
SEGMENT MASK MLAB01,CLEAR=Y.      Define a mask for this segment
    M(02,33) 'USED BODY PARTS'.   Define constant background text
    M(04,43) 'INVOICE #'.           (field titles)
    M(05,11) 'DATE:   /  /'.           *
    M(05,47) 'TERMS:'.                 *
    M(06,08) 'SOLD TO'.                *
    M(10,08) 'PART #'.                 *
    M(10,21) 'QTY'.                    *
    M(10,36) 'DESCRIPTION'.            *
    M(10,69) 'PRICE'.                  *
    M(18,55) 'SUBTOTAL  '.             *
    M(20,60) 'TAX  '.                  *
    M(22,54) 'TOTAL AMT  '.            *
    DISPLAY GR=Y.                 Enter the virtual graphics mode.
    M(11,01)
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
 .
    M(11,73) 'AAAAAAA'.
    M(12,01) 'A'.
    M(12,16) 'I'.
    M(12,25) 'I'.
    M(12,65) 'A'.
    M(12,79) 'A'.
    M(13,01) 'A'.
    M(13,16) 'I'.
    M(13,25) 'I'.
    M(13,65) 'A'.
    M(13,79) 'A'.
    M(14,01) 'A'.
    M(14,16) 'I'.
    M(14,25) 'I'.
    M(14,65) 'A'.
    M(14,79) 'A'.
    M(15,01) 'A'.
    M(15,16) 'I'.
    M(15,25) 'I'.
    M(15,65) 'A'.
    M(15,79) 'A'.
    M(16,01) 'A'.
    M(16,16) 'I'.
    M(16,25) 'I'.
    M(16,65) 'A'.
    M(16,79) 'A'.
    M(17,01)
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

```
.
    M(17,73) 'AAAAAAA'.
    M(18,65) 'A'.
    M(18,79) 'A'.
    M(19,65) 'AAAAAAAAAAAAAAA'.
    M(20,65) 'A'.
    M(20,79) 'A'.
    M(21,65) 'AAAAAAAAAAAAAAA'.
    M(22,65) 'A'.
    M(22,79) 'A'.
    M(23,65) 'AAAAAAAAAAAAAAA'.
.                                  End of segment mask block
SEGMENT SLAB01,(FLAB01),MLAB01.    Start segment SLAB01, make segment an
.                                    entry into form root FLAB01, and
.                                    use the mask MLAB01.
FIELD NVOICE.                      Define a field named NVOICE.
    POSITION(4,53)L10.             This field is positioned at line 4,
.                                     column 53 and is 10 bytes long.
    REQUIRED.                      A value must be entered into this field.
    CHAR LIST=DIGIT.               Allow only digits.
    JUST L,FILL='0'.               Justify the value to the left and fill
.                                    with zeros.
.                                   End of this field block.
FIELD MONTH.                       Define field named MONTH.
    POS(+1,17)L2.                  Line 5, column 17, 2 bytes long.
    AUTOSKIP.                      Automatically move the cursor to the
.                                    next field when this field is filled.
   REQ.
   RANGE LIST=RANGE1.             Allow digits between 1 and 12 only.
   JUST R,FILL='0'.               Justify right and fill with 0's.
.
FIELD DAYS.                         Field named DAYS.
    POS(,20)L2.                     Line 5, column 20 and 2 bytes long.
    TAB.                            Place cursor in this field if using
.                                    the Forward Tab key.
   AUTOSKIP.
   REQ.
   RANGE LIST=RANGE2.             Allow digits between 1 and 31 only.
   JUST R,FILL='0'.
.
FIELD YEAR.                         Field named YEAR.
    POS(,23)L2.                     Line 5, column 23 and 2 bytes long.
    TAB.
    REQ.
    MIN LEN=2.                      Value entered has to be at least 2 bytes.
    CHAR LIST=DIGIT.                Allow digits between 1 and 9 only.
    VALUE='81'.                     Intialize this field with 81.
.
FIELD TERMS.                        Field named TERMS.
    POS(5,53)L23.                   Line 5, column 53 and 23 bytes long.
    IF COND COND01 ON NVOICE    !   Select attributes on the basis of whether
         THEN EDITS=DOTHIS      !     the value is between 1 and 5 million
         ELSE EDITS=DOTHAT.           or its not.
```

```
.
FIELD SOLD01.                          Field named SOLD01.
    POS(+1,17)L59.                     Line 6, column 17 and 59 bytes long.
    REQ.
.
FIELD SOLD03.                          Field named SOLD03.
    POSITION (8,17)L59.                Line 8, column 17 and 59 bytes long.
    SAME AS SOLD01.                    Same attributes as the 'SOLD01' field.
.
FIELD SOLD02.                          Field named SOLD02.
    POSITION (-1,)L59.                 Line 7, column 17 and 59 bytes long.
    SAME AS SOLD01.
.
FIELD PARTN.                           Field named PARTN.
    POSITION (12,8)L5.                 Position of upper left field of array:
.                                      line 12, column 8 and 5 bytes long.
    ARRAY DIM(3,1),INC(2,1).           Three rows, one column, one line
.                                      skipped between rows, no positions
.                                      between columns.
     CHAR LIST=DIGIT.
    LENGTH LIST=LEN001.                Value must be five characters long.
.
FIELD QUANT.                           Field named QUANT.
    POS(12,20)L5.                      Line 12, column 20 and 5 bytes long.
    ARRAY DIM(3,1),INC(2,1).
    CHAR LIST=DIGIT.
    RANGE LIST=RANGE3.                 Value must be between 1 and 99999.
    JUST R,FILL=' ' ON ENTRY.          Justify immediately after value is
.                                        entered into the field.
.
FIELD DESCR.                           Field named DESCR.
    POSITION (12,26)L38.
    ARRAY DIM(3,1),INC(2,1).
    TABLE LIST=TABLE1.
    DISPLAY MASK MASK01, POSTCLEAR.  Display field mask MASK01
.
FIELD PRICE.                           Field named PRICE.
    POSITION (12,68)L8.                Line 12, column 68, 8 bytes long.
    ARRAY DIM(3,1),INC(2,1).
    CHAR LIST=MONEY.                   0 through 9,+, and .,
.                                      are the only valid chars.
    JUST R,FILL=' ',DEC=2 ON ENTRY.
    JUST R,FILL='0' ON OUTPUT.
    SCALE R,02 ON OUTPUT.
.
EXTERNAL SUBTOT.            Allow field SUBTOT to be available to the
.                           application.
FIELD SUBTOT.                          Field named SUBTOT
    POSITION (18,68)L8.                Line 18, column 68, 8 bytes long
    OUTPUT.                            Cannot enter data to this field.
.
EXTERNAL TAX.
.
FIELD TAX.                             Field named TAX.
    POSITION (20,68)L8.
    OUTPUT.
```

```
.
EXTERNAL TOTAMT.
.
FIELD TOTAMT.                       Field named TOTAMT.
    POSITION (22,68)L8.
    OUTPUT.
.
LIST CHAR DIGIT=0..9,BLANK.
.
LIST CHAR MONEY=0..9,'.',',',BLANK.
.
LIST RANGE RANGE1=IN,1/12.
.
LIST RANGE RANGE2=IN,1/31.
.
LIST RANGE RANGE3=IN,1/99999.
.
LIST RANGE RANGE4=IN,1/5000000.
.
LIST LEN LEN001=5.
.
LIST TABLE TABLE1=IN,'LEGS','EYEBALLS','FINGERS','HEARTS',
                  'KIDNEYS'.
.
CONDITION COND01.                   Condition named COND01.
    RANGE LIST=RANGE4.
.
EDIT SET DOTHIS.                    Edit set named DOTHIS.
END EDIT SET DOTHIS.                Blank edit set.
.
EDIT SET DOTHAT.                    Edit set named DOTHAT.
    REQ.
.
GROUP GROUP1=NVOICE,DAYS,MONTH,YEAR,TERMS,SOLD01,SOLD02,
             SOLD03.
.
GROUP GROUP2=PARTN(*,*),QUANT(*,*),DESCR(*,*),PRICE(*,*).
.
GROUP GROUP3=SUBTOT,TAX,TOTAMT.
.
FIELD MASK MASK01, CLEAR=N.         Field mask named MASK01.
    M(22,1) 'INVENTORY ITEMS INCLUDE:'.
    M(23,1) 'LEGS, HEARTS, FINGERS, EYEBALLS, KIDNEYS'.
.                                   End of segment SLAB01.
```

EXAMPLE

```
SEGMENT MASK OGRPSM,CLEAR=Y.  Define segment mask OGRPSM.
    M(01,10) 'ORDERED GROUP TEST: GROUP NAME IS OGRP'.
.
SEGMENT OGRPS,(OGRPF),OGRPSM. Define segment OGRPS
.
FIELD AA11.                   Define numerous field at random locations
    POS (1,1)L1.              on the screen.  All fields being of
FIELD AA310.                  1 byte long. Read the specified ordered
    POS (3,10)L1.             group names below to test for ordered
FIELD AA620.                  groups.
    POS (6,20)L1.
FIELD AA930.
    POS (9,30)L1.
FIELD AA1240.
    POS(12,40)L1.
FIELD AA1550.
    POS (15,50)L1.
FIELD AA1860.
    POS (18,60)L1.
FIELD AA2170.
    POS (21,70)L1.
FIELD AA2480.
    POS (24,80)L1.
FIELD AA180.
    POS (1,80)L1.
FIELD AA370.
    POS (3,70)L1.
FIELD AA660.
    POS (6,60)L1.
FIELD AA950.
    POS (9,50)L1.
FIELD AA1530.
    POS (15,30)L1.
FIELD AA1820.
    POS (18,20)L1.
FIELD AA2110.
    POS (21,10)L1.
FIELD AA241.
    POS (24,1)L1.
ORDER GROUP LSPIRL = AA11,AA241,AA2480,AA180,AA310,AA2110,AA2170,
                     AA370,AA620,AA1820,AA1860,AA660,AA930,AA1530,
                     AA1550,AA950,AA1240.
.
ORDER GROUP IOSPRL = AA11,AA241,AA2480,AA180,AA310,AA2110,AA2170,
                     AA370,AA620,AA1820,AA1860,AA660,AA930,AA1530,
                     AA1550,AA950,AA1240,AA950,AA1550,AA1530,AA930,
                     AA660,AA1860,AA1820,AA620,AA370,AA2170,AA2110,
                     AA310,AA180,AA2480,AA241,AA11.
.
ORDER GROUP PONG = AA11,AA2480,AA310,AA2170,AA620,AA1860,AA930,
                   AA1550,AA1240,AA1530,AA950,AA1820,AA660,AA2110,
                   AA370,AA241,AA180.
```

Appendix E

Graphic Characters

This appendix contains information on the graphic characters available on the 911 VDT, 931 VDT, Business System terminal, and 940 EVT. Figure E-1 shows the keyboard positions for the graphic characters available on the 911 VDT. Figure E-2 shows the keyboard positions for the graphic characters available on the 931 VDT. The keys that produce the graphic characters on the 940 EVT and Business System terminal are the same as the keys that produce graphic characters on the 931 VDT. Figure E-3 defines the VDT graphic character sets.

When displaying data to a field mask with the graphics display attribute, the keys with the ASCII codes specified in Figure E-3 are translated into the corresponding graphic character. All other characters are translated to a space. For information on the ASCII codes that the keys on each terminal produce, refer to either the *DX10 Operating System Application Programming Guide (Volume III)* or the *DNOS Supervisor Call (SVC) Reference Manual*, depending on the operating system you are using.

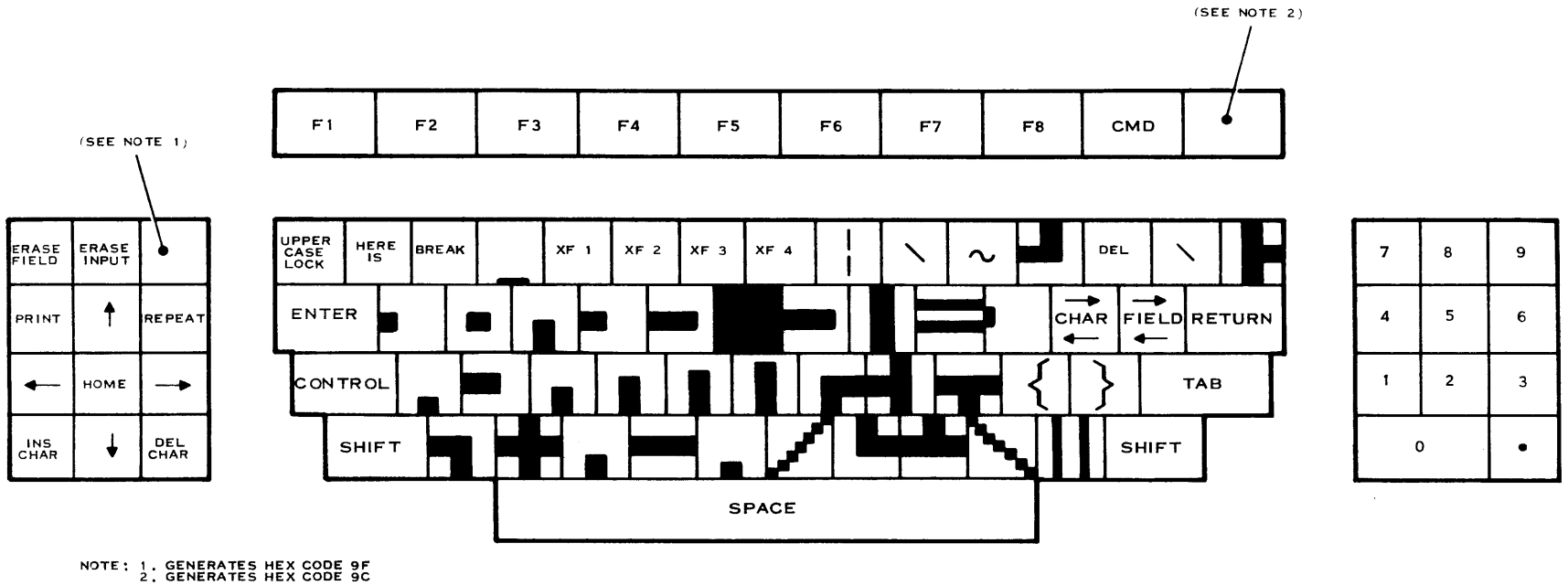


Figure E-1. Graphic Characters on the 911 VDT

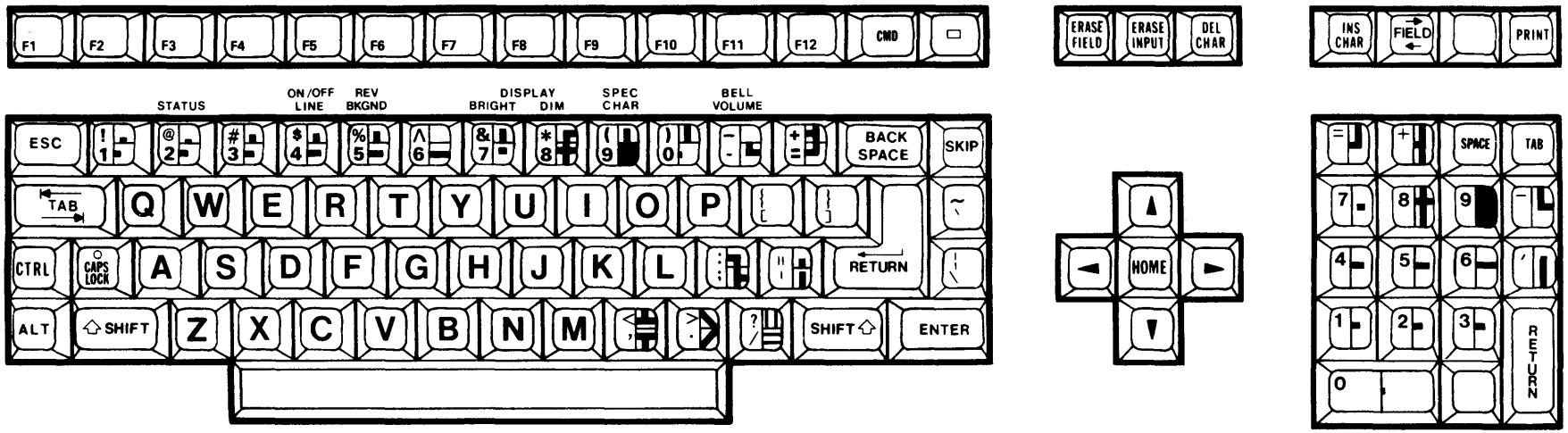


Figure E-2. Graphic Characters on the 931 VDT

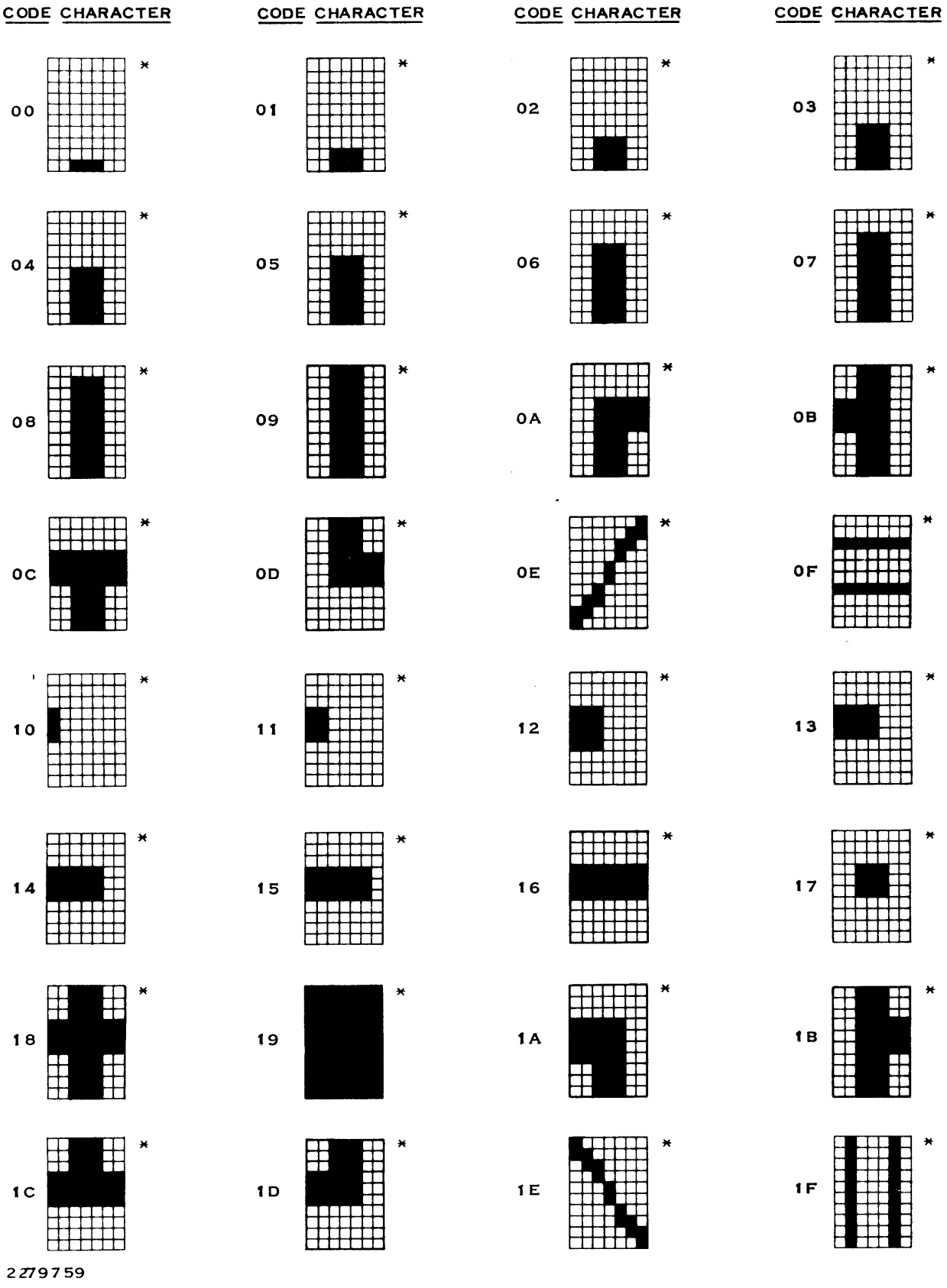


Figure E-3. VDT Graphic Character Sets

Appendix F

Quick Reference to FDL Syntax

Form Definition Language (FDL) syntax definitions use the following notation:

Notation	Meaning
Uppercase	Keywords that must be entered as shown. You can omit the rightmost letters shown in italics.
Lowercase	Generic terms (listed below) that represent the names, literals, and numbers used in your application: • A name consists of a letter followed by up to five additional letters, numerals, dashes, and dollar signs. • A literal consists of up to 78 characters enclosed by single quotes. You can use two single quotes to represent a single quote within the text. • A number is a signed integer, unless stated otherwise.
Braces	Enclose lists of items from which you choose one. Vertical bars separate items on the list.
Brackets	Enclose lists of items from which you choose one or none. Vertical bars separate items on the list.
Three dots	Mean you can enter the preceding item more than once, using commas as separators.
Punctuation	Must be entered as shown (other than the three dots): = , . " * ;
Comments	Can be included in the FDL source in four ways: • Each FDL statement ends with a period followed by a space. Any text to the right of the period is a comment. • Any text to the right of an exclamation point is a comment (unless the exclamation point is part of a literal). • Any text on a line beginning with a period is a comment. • Any text on a line beginning with a slash is a comment to be printed on a new page.

Context: Form Block

```
FORM form [,part#][,rev#]
   DISABLE CONTROL MODE mode...
   ENABLE CONTROL MODE mode...
   DEVICE = type
   FILLER = char [,DISPLAY = GRAPHICS]
      segment-mask-block
      segment-block
END FORM form
```

Context: Segment Mask Block

```
SEGMENT MASK segmask, CLEAR = {YES | NO}
   DISPLAY {ND | BR | GR} = {YES | NO}...
   M([row],[col]) literal
END SEGMENT MASK segmask
```

Context: Segment Block

```
SEGMENT segment [,(form...)] [,segmask]
   DEVICE = type
   ERROR MESSAGE message = literal
   EXTERNAL {field | variable}...
   FKEYS key/iofield...
   FILLER = char [,DISPLAY = GRAPHICS]
   GROUP group = {field | variable | group | array}...
   LIST CHARACTER list = {char | char..char | char/char}...
   LIST LENGTH list = length...
   LIST RANGE list = {IN, | EX,} {value..value | value/value}...
   LIST SUBSTITUTE list = value/literal...
   LIST TABLE list = {IN, | EX,} value...
      condition-block
      edit-set-block
      field-block
      field-mask-block
   M([row],[col]) literal
   ORDERED GROUP group = {field | variable | group | array}...
   VARIABLE variable = literal
END SEGMENT segment
```

Context: Field Block

```
FIELD field
   ARRAY DIMENSION(row,col), INCREMENT(offset,offset)
   AUTOSKIP
   BRANCH TO iofield
   CHARACTER LIST = list [;DIAGNOSTIC = {message | literal}]
   COPY FROM {field | variable | literal | *}... TO {field | variable | *}... ON ENTRY
   COPY FROM {field | variable | literal | *}... TO * ON COMPLETION
   DEFAULT = {iofield | variable | literal}
   DISPLAY {ND | BR | BL | GR} = {YES | NO}...
   DISPLAY MASK mask [,POSTCLEAR]
   EXTERNAL {field | variable}...
   FILLER = char [,DISPLAY = GRAPHICS]
   GRAPHICS INPUT
   [PASS | FAIL] IF {iofield | variable | *} relop {iofield | variable | *}
      [;DIAGNOSTIC = {message | literal}]
   IF [NOT] CONDITION condition [PREENTRY | POSTENTRY] ON {iofield | variable | *}
      THEN {GOTO iofield | TERMINATE READ [IMMEDIATELY] | EDITS = edits}
      [ELSE {GOTO iofield | TERMINATE READ [IMMEDIATELY] | EDITS = edits}]
   JUSTIFY LEFT, FILLER = char [ON ENTRY | ON COMPLETION]
   JUSTIFY RIGHT, FILLER = char [,DECIMAL = places] [ON ENTRY | ON COMPLETION]
   LENGTH LIST = list [;DIAGNOSTIC = {message | literal}]
   MINIMUM LENGTH = length [;DIAGNOSTIC = {message | literal}]
   NOAUTOSKIP
   NO ENTRY
   NOTAB
   NOTREQUIRED
   NOVALIDATE
   [SIGNED | UNSIGNED] NUMERIC [,FILL = char] [,DECIMAL = places]
   OUTPUT
   POSITION [R] (row,col) L length
   PROMPT = literal
   RANGE LIST = list [;DIAGNOSTIC = {message | literal}]
   REQUIRED [;DIAGNOSTIC = {message | literal}]
   SAME AS {iofield | edits | *} [EXCEPT FOR]
   SCALE {L | R}, places [ON ENTRY | ON COMPLETION]
   SUBSTITUTE LIST = list [ON ENTRY | ON COMPLETION]
   TAB
   TABLE LIST = list [;DIAGNOSTIC = {message | literal}]
   TERMINATE READ [IMMEDIATELY]
   VALUE = {iofield | variable | literal}
END FIELD field
```

Context: Field Mask Block

```
FIELD MASK mask, CLEAR = {YES | NO}
  DISPLAY {ND | BR | GR} = {YES | NO}...
  M([row],[col]) literal
END FIELD MASK mask
```

Context: Edit Set Block

```
{EDIT SET | EDITS} edits
  AUTOSKIP
  BRANCH TO iofield
  CHARACTER LIST = list [;DIAGNOSTIC = {message | literal}]
  COPY FROM {field | variable | literal | *}... TO {field | variable | *}... [ON ENTRY]
  COPY FROM {field | variable | literal | *}... TO * ON COMPLETION
  DISPLAY {ND | BR | BL | GR} = {YES | NO}...
  DISPLAY MASK mask [,POSTCLEAR]
  FILLER = char [,DISPLAY = GRAPHICS]
  GRAPHICS INPUT
  [PASS | FAIL] IF {iofield | variable | *} relop {iofield | variable | *}
    [;DIAGNOSTIC = {message | literal}]
  IF [NOT] CONDITION condition [PREENTRY | POSTENTRY] ON {iofield | variable | *}
    THEN {GOTO iofield | TERMINATE READ [IMMEDIATELY] | EDITS = edit}
    [ELSE {GOTO iofield | TERMINATE READ [IMMEDIATELY] | EDITS = edit}]
  JUSTIFY LEFT, FILLER = char [ON ENTRY | ON COMPLETION]
  JUSTIFY RIGHT, FILLER = char [,DECIMAL = places] [ON ENTRY | ON COMPLETION]
  LENGTH LIST = list [;DIAGNOSTIC = {message | literal}]
  MINIMUM LENGTH = length [;DIAGNOSTIC = {message | literal}]
  NOAUTOSKIP
  NO ENTRY
  NOTAB
  NOTREQUIRED
  NOVALIDATE
  [SIGNED | UNSIGNED] NUMERIC [,FILL = char] [,DECIMAL = places]
  PROMPT = literal
  RANGE LIST = list [;DIAGNOSTIC = {message | literal}]
  REQUIRED [;DIAGNOSTIC = {message | literal}]
  SAME AS {iofield | edits | *} [EXCEPT FOR]
  SCALE {L | R},places [ON ENTRY | ON COMPLETION]
  SUBSTITUTE LIST = list [ON ENTRY | ON COMPLETION]
  TAB
  TABLE LIST = list [;DIAGNOSTIC = {message | literal}]
  TERMINATE READ [IMMEDIATELY]
  VALUE = {iofield | variable | literal}
END EDIT SET edits
```

Context: Condition Block

```
COND/TION condition
   CHARACTER LIST = list [;DIAGNOSTIC = {message | literal}]
   [PASS | FAIL] IF {iofield | variable | *} relop {iofield | variable | *}
      [;DIAGNOSTIC = {message | literal}
   LENGTH LIST = list [;DIAGNOSTIC = {message | literal}]
   MIN/MUM LENGTH = length [;DIAGNOSTIC = {message | literal}]
   RANGE LIST = list [;DIAGNOSTIC = {message | literal}]
   TABLE LIST = list [;DIAGNOSTIC = {message | literal}]
END COND/TION condition
```

Definition of Generic Terms

Term	Meaning
array	Array element: Field name followed by row and column
char	Character: Alphanumeric character enclosed in single quotes
col	Column number or offset: Unsigned for absolute column number, signed for offset from previous column specification
condition	Condition name: Unique within its segment
edits	Edit set name: Unique within its segment
field	I/O or output field name: Unique within its segment
form	Form name: Unique within its program file
group	Group name: Unique within its segment
iofield	I/O field name: Unique within its segment
key	Function key: Device-specific code for a function key
length	Field length: Positive integer
list	List name: Unique within its segment
literal	Text string: Up to 78 characters enclosed in single quotes
mask	Field mask name: Unique within its segment
message	Message name: Unique within its segment
mode	Control mode: See Table 3-1
offset	Row/Column offset: Positive integer
part#	Part number: Six-digit integer
places	Decimal places: Positive integer
relop	Relational operator: EQ, NE, LT, LE, GT, or GE
rev#	Revision number: Two-digit integer
row	Row number or offset: Unsigned for absolute row number, signed for offset from previous row specification
segmask	Segment mask name: Unique within its program file
segment	Segment name: Unique within its program file
type	Device type: See Table 3-2
value	Value: Possible user response
variable	Variable name: Unique within its segment

Appendix G

Quick Reference to the ISGE

G.1 INTRODUCTION

This appendix provides a quick reference for the ISGE. Figure G-1 shows the three major phases in an ISGE session: initiation, design, and termination. This appendix concentrates on information you need for the design phase. It describes the function keys you can use when designing a segment mask or a field mask. Then, it describes each of the prompts on the Field Attribute Selection (FAS) menu and the Edit Set Specification (ESS) menu.

G.2 SEGMENT MASK AND FIELD MASK DESIGN

Table G-1 describes the active function keys you can use when designing a segment mask or field mask.

Table G-1. Function Keys

Function	Generic Key Name
Position Cursor	F1
Insert Lines	F2
Delete Lines	F3
Draw Vertical	F4
Copy Block	F5
Move Field	F6
Delete Field	F7
Insert Field	F8
Enter/Leave Field Mask Mode	F10
Leave Edit Screen Mode or Abort	Command
Print	Print

G.3 FIELD ATTRIBUTE AND EDIT SET SPECIFICATION

The following list explains the prompts on the Field Attribute Specification (FAS) menu and the Edit Set Specification (ESS) menu. It describes the prompts in the order they appear on the screens. The differences between the two menus are noted. Figure G-2 shows the FAS menu, and Figure G-3 shows the ESS menu.

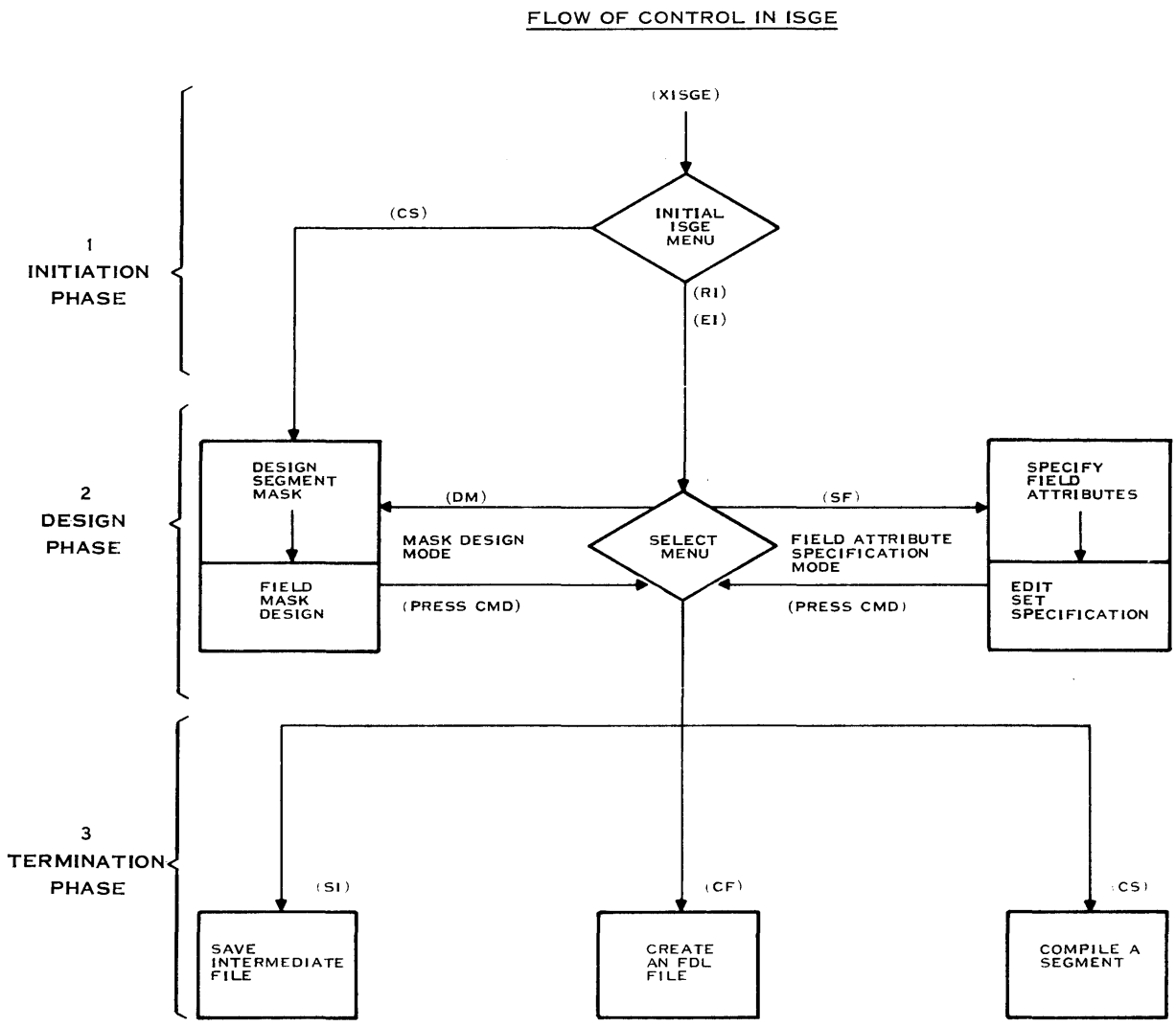


Figure G-1. ISGE Flow of Control

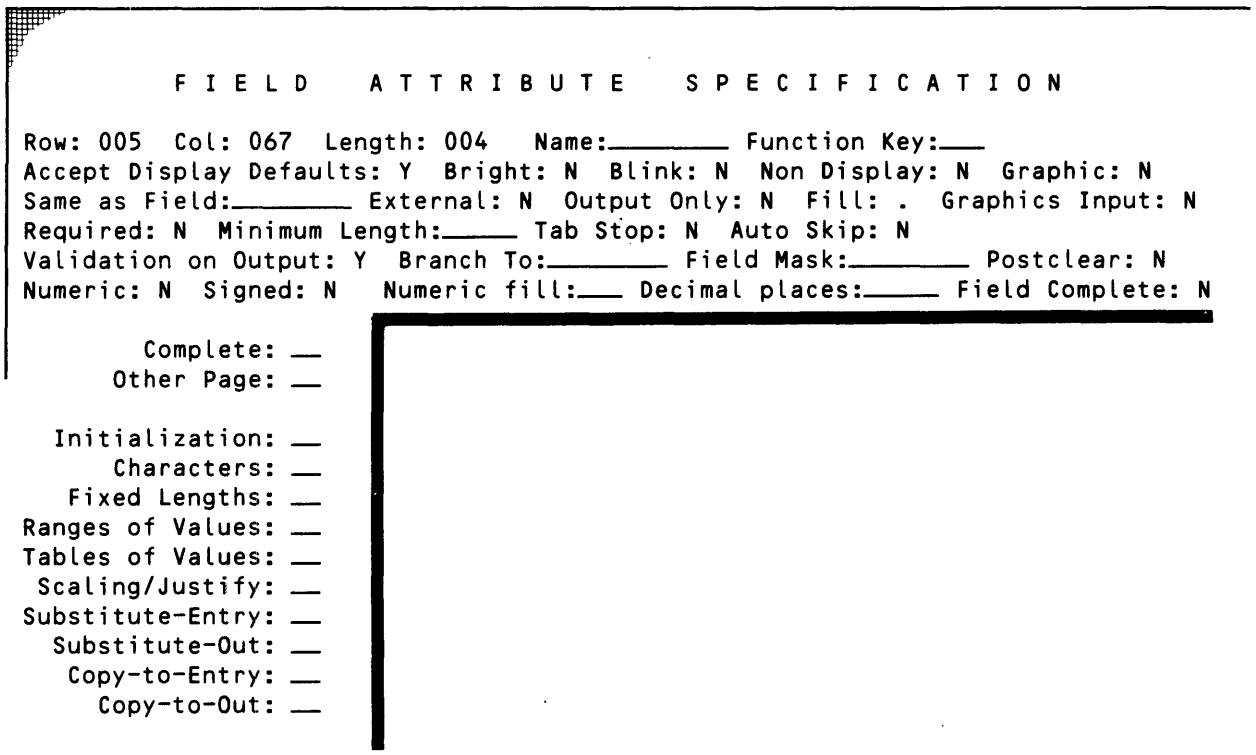


Figure G-2. FAS Menu

The prompts from Row to Output Only appear only on the FAS menu.

Row and Column — This attribute indicates the line and column on the segment mask of the first character of the field.

Length — This attribute indicates the length of the field.

Name — This attribute indicates the name of the field. It is an optional attribute. If you do not specify a name, a default name is created.

Function Key — This attribute associates a specific function key with the field so that pressing the function key causes an immediate branch to the specified field.

Accept Display Defaults — This attribute allows you to specify whether you want to accept the default values listed for the four display attributes that follow. They are bright, blink, non-display, and graphics. Enter Y here to accept the defaults for these attributes and to move the cursor to the next line in the menu; enter N to select any of these four attributes.

Bright — This display attribute specifies that information entered in this field will be brighter than the rest of the segment mask.

Blink — To the extent that this attribute is supported by the device, the entities in the field will blink.

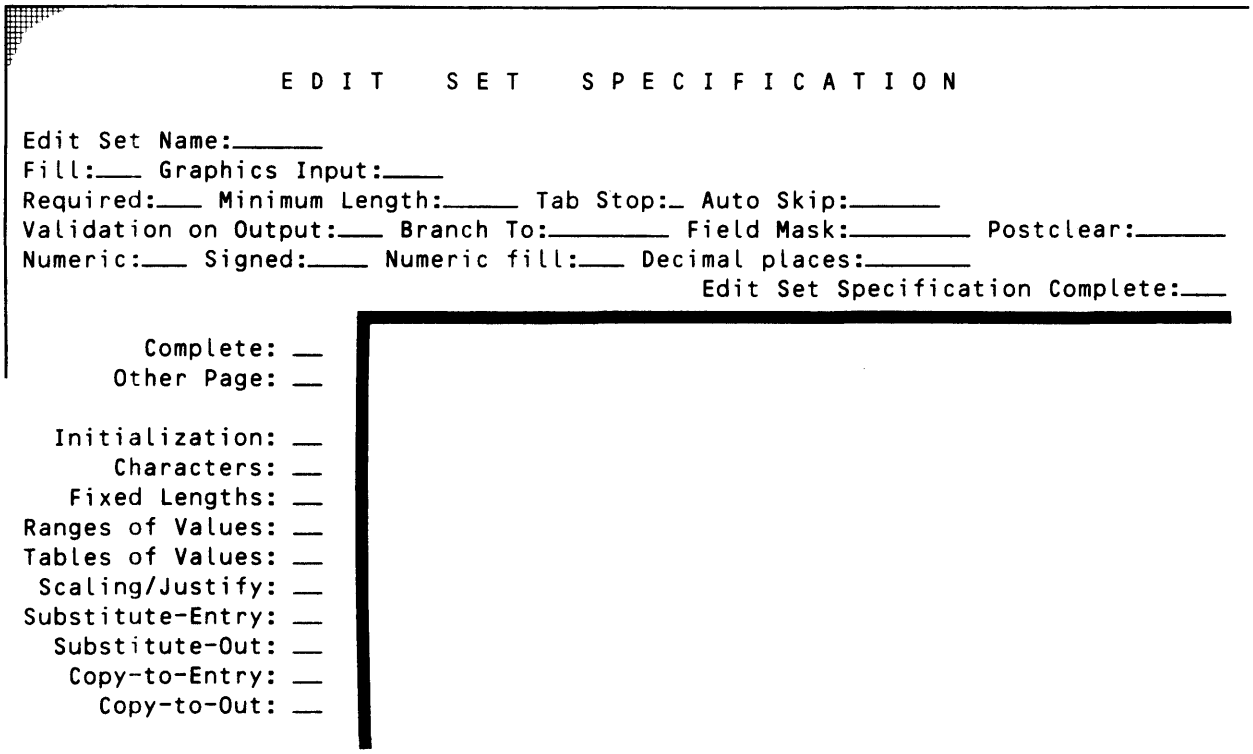


Figure G-3. ESS Menu

Non Display — This display attribute specifies that information entered in this field will not be displayed.

Graphic — This display attribute specifies that graphics characters can be written to the field.

Same as Field — This attribute specifies that the current field has the same attributes as the field you name here except for the position and display attributes. If you specify other attributes, they are discarded and a warning message is displayed.

External — This attribute specifies that the name of the field is external. If a field name is external, the application can refer to it by name in Read, Write, and Reset commands.

Output Only — This attribute prohibits user input.

The following prompt appears only on the ESS menu.

Edit Set Name — This prompt specifies the name of the edit set being defined.

The remainder of the prompts are identical on both menus.

Fill — This attribute allows you to specify a character that will be used to represent empty field positions on the screen. The default fill character is the underscore (_).

Graphics Input — This attribute specifies that terminal-dependent graphics characters can be entered from the keyboard into the field.

Required — This attribute specifies that data must be entered in this field. If a required field is left empty or filled with blanks, an error message is displayed.

Minimum Length — This attribute specifies a minimum length for data entered in this field.

Tab Stop — This attribute specifies that the cursor stops in this field when the Forward Tab key is pressed.

Auto Skip — This attribute specifies that the cursor automatically goes to the next field when the last character position in this field is filled.

Validation on Output — This attribute specifies that the field is not validated immediately prior to the return of data to the application.

Branch To — This attribute names the next field to read. The cursor goes to the named field rather than the next field in normal sequence.

Field Mask — This attribute names a field mask to display when the cursor is in this field.

Postclear — This attribute specifies that the field mask is cleared from the screen when the cursor leaves the field.

Numeric — This attribute specifies that only numbers can be entered in this field.

Signed — This attribute specifies that signed numbers can be entered in this field.

Numeric Fill — This attribute allows you to specify the character that will be used to fill spaces not filled by a number in a field with the attribute Numeric.

Decimal Places — This attribute specifies the number of decimal places for numbers entered into the field.

Field Complete — Enter an N here to continue specifying attributes for the field; enter a Y if attribute specification for this field is complete.

Complete — Press the F1 key to return to the field labeled Field Complete.

Other Page — Not all the attributes for this section of the menu can be displayed at one time. Press the F1 key to see additional attributes. Figure G-4 shows the FAS screen when you press F1 for Other Page. The ESS screen is identical for these attributes.

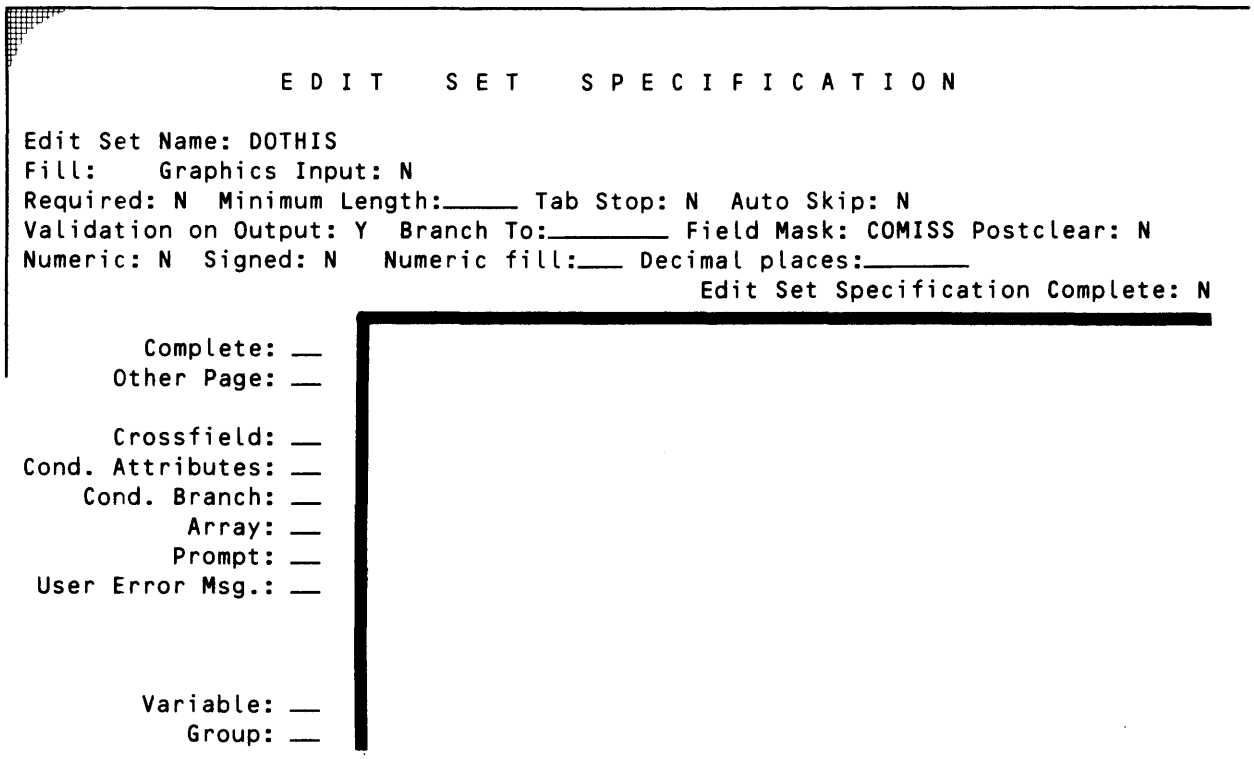


Figure G-4. Other Page of FAS Menu

To specify any of the following attributes, you need to position the cursor in the field and press F1. This displays the prompts for that attribute, shows the syntax for the FDL statements generated by this specification, and provides an example of an FDL statement that specifies this attribute. Figure G-5 shows the screen when you press F1 for the Characters prompt. The other prompts display screens similar to this one when you press F1.

Initialization — This attribute specifies default or initial values for this field.

Characters — This attribute specifies a list of valid characters for this field.

Fixed Lengths — This attribute specifies a list of valid input data lengths for this field.

Ranges of Values — This attribute specifies a list of valid ranges of values for this field.

Tables of Values — This attribute specifies a list of valid tables of values for this field.

Scaling/Justify — This attribute specifies scaling and/or justification for this field.

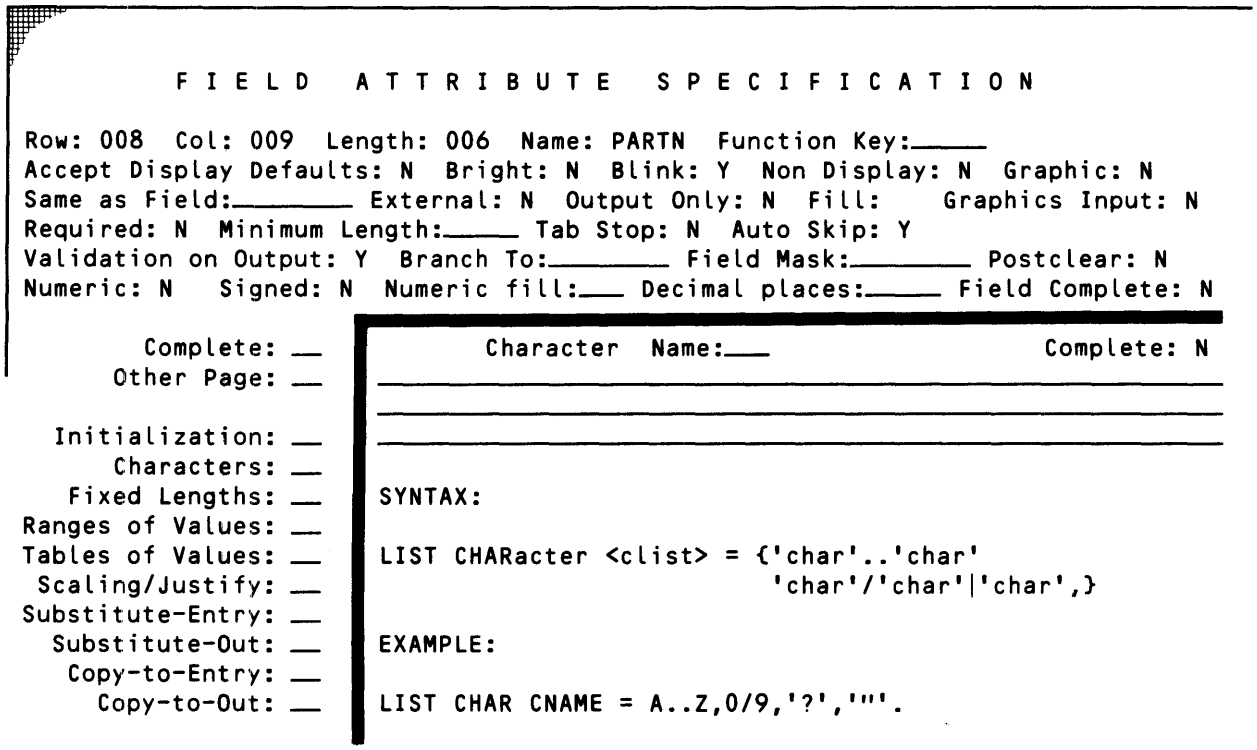


Figure G-5. FAS Menu — Character Prompt

Substitute-Entry — This attribute specifies a list of substitutions to be made on entry into this field.

Substitute-Out — This attribute specifies a list of substitutions to be made on return to the application.

Copy-to-Entry — This attribute names the fields to which input data is to be copied on data entry.

Copy-to-Out — This attribute names the fields to which input data is to be copied on return to the application.

Crossfield — This attribute specifies crossfield comparison edit for the field. The two fields named here are compared to see if they fulfill one of the following conditions: equal (EQ), less than (LT), less or equal (LE), not equal (NE), greater than (GT), or greater or equal (GE).

Cond.Attributes — This attribute specifies conditional attributes for the field. If the stated condition is true, one edit set is selected; if the condition is false, a different edit set is selected.

Cond.Branch — This attribute specifies conditional field branching. If the condition specified is true, the cursor goes to the field specified rather than to the next field in normal sequence.

Array — This attribute specifies replication of the field into an array.

Prompt — This attribute applies only to KSR-type devices. If specified when any other device is being used, the prompt is ignored during form execution. This attribute specifies KSR unformatted input prompt text.

User Error Msg — This attribute specifies custom edit error messages.

Variable — This attribute specifies a variable for this field.

Group — This attribute specifies a group for this segment.

Glossary

Array — A collection of fields arranged in rows and columns. All fields in an array have the same attributes, except for location.

Attribute — See Field Attribute.

Auto Skip — An attribute of a field that specifies the cursor automatically goes to the next field after the user enters a character into the last position in the field.

Block — A group of FDL statements that appear together and define one of the components of a form. FDL contains seven types of blocks, one for each of the components of a form: form, segment, segment mask, field, edit set, condition, and field mask. The block defines the structure of, and attaches a name to, these components of the form.

Branch — An attribute of a field that names the next field to read. The cursor goes to the named field rather than the next field in the usual order.

Character Set — An attribute of a field that determines the set of valid characters the user can enter into the field.

Condition Block — A series of FDL statements that define a condition. You can define conditions according to the character composition, length, or value of the data at the time the user enters data into the field or before the data is returned to your program. The condition can determine which edit set to apply or where to move the cursor for the next operation. The CONDITION statement marks the beginning of the condition block and assigns it a name.

Control Mode — One of the 13 methods of operation that your application can specify. Refer to Table 3-1.

Copy — An attribute of a field that causes a specified field, value, or literal to be copied to another field or variable after the user enters data into the field that has the attribute. Though the field with the copy attribute is usually the one copied, this is not a requirement.

Data Buffer — A data structure in the application program used to exchange information with the Form Executor.

Deblank — The substitution of ASCII nulls for blanks.

Decimal Places — An attribute of a field that specifies the number of decimal places for numbers entered into the field.

Default Value — An attribute of a field that specifies the field automatically receives a value at the time the segment is displayed. If the cursor never enters the field, this value is also used in post-processing. The default value is different from an initial value in that the cursor must enter the field for an initial value to come into effect and in that the initial value is restored each time the cursor returns. The default value is only displayed once—when the Prepare Segment routine is called. See also Initial Value.

Delayed Write Mode — A mode of operation unique to KSR-type terminals in which information passed to the terminal by the use of a Write command from the application is stored in the virtual screen maintained by the Form Executor, but is not printed. The information written to the fields of the group is printed only as a result of a subsequent Read command or an action of the terminal user, such as pressing the Print key. See also Immediate Write Mode.

Device Type — The generic device where the form is displayed. TIFORM supports several types of VDTs and KSRs.

Display — An attribute of a field that determines how values in the field appear on the user's screen. TIFORM can support the following display attributes: bright, blink, graphics, reverse, and nondisplay.

Edit Key — A key on the terminal that allows the user to perform editing functions such as moving the cursor, erasing input, and closing a read. The functions and effects of edit keys are determined by the operating system and the Form Executor. See also Function Key.

Edit Set — A set of field attributes that determine how a field is displayed to the user, which kinds of data it accepts, and what processing follows data entry.

Edit Set Block — A series of FDL statements that define the attributes in an edit set. The EDIT SET statement marks the beginning of the edit set block and assigns it a name.

Elemental Membership — The set of fields, arrays, and variables that belong to a group. To derive the elemental group from its FDL, you replace each subgroup with the items it contains until only fields, arrays, and variables remain.

Event Key — A key used to signal an event rather than enter data. See also Function Key.

Exact Length — An attribute of a field that specifies the user must enter a specific number of characters when entering data into the field or the entry is rejected.

External Name — A name that your application program can use to reference a field or variable in your form. You declare a field or variable external by specifying it in an EXTERNAL statement.

FDL — See Form Definition Language.

FDL Compiler — The software component of TIFORM that translates your FDL statements into a format usable by the Form Executor and installs them as overlays in a specified program file.

FDLC — See FDL Compiler.

Field — A part of a segment used for I/O between the application program and the user. Each field occupies one or more consecutive horizontal positions on a segment, as determined by FDL statements for the field. Using calls to the interface routines, the application program can read and write information in the fields for the currently active segment. *Output only* fields can only be written by the application program. If read, they return blanks. *Input/Output* or *I/O* fields can be both written and read.

Field Attribute — A characteristic of a field, such as its length, location, or usage.

Field Block — A series of FDL statements that define a field and its attributes. The FIELD statement marks the beginning of a field block.

Field Mask — Background text that is displayed when the cursor enters the field. Field masks are often used to display instructions about a particular field but are not used for data entry.

Fill Character — A character that replaces the blank as the filler for input fields on the form.

Fixed Length — See Exact Length.

Form — (1) To the user of a TIFORM application, a sequence of screen displays used to enter data for the application. (2) To the application program, a collection of segments, previously defined in FDL and stored together in a program file as overlays.

Form Block — A series of FDL statements that define a form. The FORM statement marks the beginning of a form block.

Form Definition Language (FDL) — The language that allows you to design flexible, attractive forms for data entry. FDL is a non-procedural, block-structured language that you can use to specify the characteristics of your forms. The FDL compiler translates your FDL statements into a format usable by the Form Executor and installs them as overlays in a specified program file.

Form Designer — The person who designs a TIFORM form for an application program.

Form Executor — The TIFORM run-time associated with the terminal where your forms are displayed. The Executor receives calls from the TIFORM interface routines linked with your application program and carries out their instructions according to the characteristics of the terminal.

Form Tester — A utility that enables you to test your forms without having to write a test program. The Form Tester serves as an interactive driver program that allows you to simulate calls to the interface routines and immediately see their results. It also allows you to delete form segments, segment masks, and roots from a form program file.

Formatted Input — A mode of operation unique to KSR-type terminals in which the entire KSR screen is available for printing. See also Unformatted Input.

Function Key — A key on the terminal, such as F1, that performs a predefined function for the user. The effects of function keys are determined by the form designer and/or the application program. See also Edit Key.

Generic Key Name — A general name for a key function that applies to all supported terminals. Appendix A describes the specific key on each terminal that performs the function.

Graphics Input — An attribute of a field that specifies the user can enter graphics characters as data.

Group — A list of named items that is assigned a group name. These named items can be fields, variables, or other groups. These items are read in the order they appear on the screen: left to right, top to bottom. See also Ordered Group.

Immediate Write Mode — A mode of operation unique to KSR-type terminals in which the contents of the fields of a group are printed immediately following the receipt of the contents by the Form Executor. See also Delayed Write Mode.

Initial Value — An attribute of a field that specifies the field automatically receives a specified value whenever it becomes the current field. As the cursor enters the field, the initial value is displayed, making it easy for the user to enter that value by simply tabbing through the field. If you do not assign an initial value to the field, its current value is displayed whenever the cursor enters the field.

Input/Output (I/O) Field — A field that can be both written and read by the application program. For instance, an application program can first write a dollar sign into a monetary I/O field and later read the user's response from the same field.

Interactive Screen Generator/Editor (ISGE) — A utility program that helps you design forms for your application. Using ISGE, you can create a new segment (screen display) and modify it until you are satisfied with the layout. ISGE then translates your design into FDL statements and compiles the resulting FDL program. ISGE leads you through the form definition procedure with a series of menus and prompts, using the function keys on your terminal for many common operations.

Interface Routines — Subprograms that allow your application program to access the forms you create.

Intermediate Segment File — During execution of the ISGE, the segment being operated on is maintained in a format unique to the ISGE. The file containing a segment in this format is referred to as an intermediate segment (IMS) file, and the segment is called an intermediate segment.

Interprocess Communication (IPC) — The DNOS capability that permits two or more tasks to exchange information.

Intertask Channel Clearer — A DX10 utility that clears intertask communication (ITC) channels.

Intertask Communication (ITC) — A DX10 facility that permits the establishment of communication channels and the exchange of data between two tasks.

I/O Field — See Input/Output Field.

IPC — See Interprocess Communication.

ISGE — See Interactive Screen Generator/Editor.

ITC — See Intertask Communication.

Justification — An attribute of a field that calls for the Form Executor to left or right justify data that does not fill the entire field.

Left justification: Usually used for text data where the application program expects to receive a character string of a certain length, without leading blanks but with enough trailing blanks to fill the field.

Right justification: Usually used for numeric data where the application program expects to receive a valid numeric value, possibly with a sign to the left of the first significant digit or as the rightmost character in the value.

KSR — A terminal that contains a printer and a keyboard and allows input/output from the computer.

Linkable Executor — A special version of the Form Executor that you can link directly with your application program instead of using the standard Form Executor, which runs as a separate task.

Minimum Length — An attribute of a field that specifies the user must enter at least a minimum number of characters when entering data into the field or the entry is rejected.

No Entry — An attribute of a field that allows you to display a field without permitting the user to enter data into it. Unless you assign the field a no entry or output attribute, the user is free to enter or modify data in the field. The output and no-entry attributes differ in the way the field is processed and the other attributes it can have. If you assign the no-entry attribute to a field, it remains an I/O field with all of the properties of an I/O field except that the cursor can never enter the field during a read. Post-entry processing occurs as usual, and Read routines return the appropriate values.

Novalidate — An attribute of a field that exempts it from the usual revalidation that occurs for all fields just prior to returning data to your program.

Numeric — An attribute of a field that specifies the field only accepts numeric data. The user can enter only the numerals 0 – 9 and, in some cases, blanks, a sign, and a decinal point. Numeric fields are right-justified before they are returned to the application program.

Numeric Fill — An attribute of a field that allows you to specify the character that is used to fill positions not occupied by a number in a field with the attribute Numeric.

On Completion — A time when a postentry operation is performed on the data entered into a field. The operation is not performed until the data for all fields in the current application Read is returned to your program. See also On Entry.

On Entry — A time when a postentry operation is performed on the data entered into a field. The operation is performed immediately after the user enters data into the field. See also On Completion.

Ordered Group — A list of named items that is assigned a group name. These named items can be fields, variables, or other groups. These items are read in the order specified by their FDL statements rather than in the order in which they appear on the screen. See also Group.

Output — An attribute of a field that allows you to display a field without permitting the user to enter data into it. Unless you assign the field an output or no entry attribute, the user is free to enter or modify data in the field. The output and no-entry attributes differ in the way the field is processed and the other attributes it can have. If you assign the output attribute to a field, it becomes an output field and the only other FDL statements allowed in its field block are POSITION, DISPLAY, and ARRAY. Your program can read, write, and reset the field, but any Read always returns blanks.

Output-Only Field — A field with the output attribute. An output field can only be written by the application program. The field cannot be used for data entry.

Overlay — A part of a task that resides on disk until explicitly requested. When requested, the overlay replaces part of the task previously in memory. Using overlays can reduce the amount of memory required by a task to the amount required for the largest segment requiring memory at one time.

Page — The KSR equivalent of the VDT screen.

Position — An attribute of a field that tells how many characters it can contain and where it begins on the screen or page.

Postclear — An attribute of a field that specifies the field mask is cleared from the screen when the cursor leaves the field.

Print Key — A key that you can press to print the contents of the terminal's screen when you are executing a form.

Prompt — An attribute of a field that applies to KSR-type devices only and is used to expand incomplete background text. If the device type is anything else, the prompt is ignored during form execution.

Ranges of Values — See Value Range.

Required — An attribute of a field that specifies the field cannot be left empty by the user. If a field does not have the required attribute, the user can skip it or erase any value already there.

Scaling — A specification in the numeric attribute of a field that calls for the Form Executor to multiply or divide the value of a numeric entry by a power of ten. When the user enters a number without a decimal point into a field with a *left* scaling attribute, a decimal point is inserted a specified number of positions from the right end of the field—in effect dividing the number by a power of ten. When the user enters a number without a decimal point into a field with *right* scaling, a specified number of zeros are inserted at the end of the field—in effect multiplying the number by a power of ten. If a field has both a scaling attribute and a justification attribute, the scaling occurs prior to justification.

SCI — The user interface to the operating system. It prompts the user, interprets responses as commands, and directs activities in the operating system to satisfy those commands. You can also use SCI as a programming language.

Screen — The part of a form that can be displayed on a 24-line by 80-character VDT screen or the virtual screen of a KSR. See also Page.

Segment — The part of the form displayed for data entry. The segment has fields where the user can enter data and where the program can display messages. The segment can also contain segment masks, fields, and field masks.

Segment Block — A series of FDL statements that define a segment block. The SEGMENT statement marks the beginning of a segment block.

Segment Mask — The part of the segment that consists of graphics or text that is displayed with the segment but not used for data entry. The segment mask consists of directions, prompts, borders, and other constant text.

Segment Mask Block — A series of FDL statements that define a segment mask. The SEGMENT MASK statement marks the beginning of a segment mask.

Signed — A specification of the numeric attribute of a field that specifies signed numbers can be entered in the field.

Substitute — An attribute of a field that specifies a predetermined substitute replaces the value that the user enters into the field.

System Command Interpreter — See SCI.

Tab — An attribute of a field that specifies the cursor stops at the beginning of the field when the user presses the Forward Tab key to exit the previous field.

Tables of Values — See Value Table.

Terminal User — See User.

Terminate — An attribute of a field that specifies a premature termination of the current Read. If you specify that the termination is to occur IMMEDIATELY, all data is returned to your program without the usual revalidation or on-completion processing. If you do not specify an immediate termination, revalidation and on-completion processing occur but the Read terminates before the user has a chance to enter data in any other fields.

Text Editor — A utility supplied with the operating system that allows you to create and modify files of data.

Unformatted Input — A mode of operation unique to KSR-type terminals in which only the field prompts are printed along with the current contents of fields into which the user is to enter data. See also Formatted Input.

User — The person who uses the completed application program.

Validation on Output — An attribute of a field that specifies the field is not validated immediately prior to the return of data to the application.

Value Comparison — An attribute of a field that requires data to pass a comparison test before it is accepted. The comparison test checks the relationship between the current values of two fields or variables—not necessarily the field that you assign the value comparison attribute. When the user enters data into the field, the comparison test is made. Depending on whether the data entered passes or fails the test, processing continues or the user receives an error message and tries again.

Value Range — An attribute of a field that applies only to numeric data. If a field has this attribute, it accepts only values that lie within a defined range or set of ranges. A user who attempts to enter an unacceptable value receives an error message and must try again.

Value Table — An attribute of a field that specifies the field only accepts values that appear in a table of valid values. When the user enters a number into a field with this attribute, the number is compared to a list of valid ranges assigned to the field. If the value does not appear on the table, the user receives an error message and must try again.

Variable — A storage area within a segment that can be written to and read from by an application. You can also use variables for validation tests expressed within the segment. Data contained in a variable is never displayed.

Video Display Terminal — See VDT.

Video Display Unit — See VDU.

VDT — A terminal that contains a VDU and a keyboard and allows input/output from the computer.

VDU — The part of a VDT that contains the cathode-ray tube (CRT).

Alphabetical Index

Introduction

HOW TO USE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables.

INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections — Reference to Sections of the manual appear as "Sections x" with the symbol x representing any numeric quantity.

- Appendixes — Reference to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.

- Paragraphs — Reference to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph may be found.

- Tables — References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number.

  Tx-yy

- Figures — References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number.

  Fx-yy

- Other entries in the Index — References to other entries in the index preceded by the word "See" followed by the referenced entry.

USER'S RESPONSE SHEET

Manual Title: TIFORM Reference Manual (2234391-9701)

Manual Date: February 1984 **Date of This Letter:** _____

User's Name: _____ **Telephone:** _____

Company: _____ **Office/Department:** _____

Street Address: _____

City/State/Zip Code: _____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

CUT ALONG LINE

Location in Manual	Comment/Suggestion
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP

ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769

FOLD

Cover Part No. 2310002-0001

TEXAS
INSTRUMENTS