308-388
Issue 1

# AT&T 3270 Emulator + ESCORT™

## Programmer's Guide

**©1988 AT&T**
**All Rights Reserved**
**Printed in USA**

**NOTICE**

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

DEC is a registered trademark and VT100 is a trademark of Digital Equipment Corporation.
IBM is a registered trademark of International Business Machines Corporation.
MS-DOS is a registered trademark of Microsoft Corporation.
Tektronix is a registered trademark of Tektronix, Inc.

# Contents


1 **Getting Started**

Overview 1-1
What Is In This Guide 1-3
Other ESCORT Documentation 1-9

2 **Programming in ESCORT**

Overview 2-1
Program Structure 2-3
Character Set 2-7
Reserved Words 2-9
Constants and Variables 2-11
Operators and Expressions 2-21
Local Session Screens 2-29
Special Features 2-35
Synchronous and Asynchronous
Host Programming Considerations 2-41

3 **Sample Programs**

Overview 3-1
Synchronous Host Sample Program 3-3
Asynchronous Host Sample Program 3-39

4 **Commands and Functions**

How to Use This Section 4-1
Command Directory 4-3
Function Directory 4-137

## 5 ESCORT Utilities

Overview 5-1
Upload and Download 5-3
Generating Screen Field Variables 5-17
Get Fields 5-25
Asynchronous Host Soft Function Keys 5-29

## 6 Local Screen Generator Utility Program

Overview 6-1
Accessing and Quitting LSGEN 6-3
Creating and Editing Fields 6-9
Defining Fields 6-19
LSGEN Error Messages 6-27
LSGEN Key Sequences 6-29

## Appendix A: Error Messages A-1

## Appendix B: Debugging Facilities B-1

## Appendix C: AID Subroutines Library C-1

## Appendix D: Interpretation of Attribute Bytes D-1

## Appendix E: Key Sequences E-1

## Appendix F: Environment Variables and Customization F-1

## Appendix G: Additional Programs G-1

Glossary

Index

# 1 Getting Started


**Overview** 1-1

**What Is In This Guide** 1-3
Organization of This Guide 1-3
Conventions 1-5

**Other ESCORT Documentation** 1-9

# Overview

This programmer's guide contains the information you need to know to program in ESCORT™.

The guide assumes that you have some previous programming experience and, for example, that you know how to use an IF . . . THEN statement or a WHILE . . . DO loop. It does not contain basic instructions on how to program, rather, this programmer's guide describes the rules and conventions that are particular to the ESCORT language.

Before you begin programming in ESCORT, you should have an understanding of

- the rules and requirements of the language
- the methods of constructing local session screens
- the ESCORT system global variables
- the method of handling multiple transmissions from a synchronous host
- the initialization of asynchronous communication ports
- the synchronization of data transmissions to an asynchronous host.

# What Is In This Guide

## Organization of This Guide

This guide is divided into nine parts:

### Getting Started

contains information about this programmer's guide. Read this chapter to learn how the guide is organized, and what conventions and definitions are used throughout the guide.

### Programming in ESCORT

covers the rules and conventions of the ESCORT language. Read this chapter to learn how to structure an ESCORT program and what types of variables, operators, and expressions are permissible in this language. Special features of the ESCORT language are also presented here.

This chapter contains information on how to handle specific host system situations, such as partial system responses in the synchronous environment, the initialization of asynchronous communication ports, and the synchronization of data transmission to and from an asynchronous host.

### Sample Programs

demonstrates how ESCORT works. Read this chapter to understand ESCORT program structures and how to execute programs in both the synchronous and asynchronous environments.

### Commands and Functions

presents an alphabetic listing of all ESCORT commands and functions. Use this chapter as a reference manual to look up the correct format of all commands and functions. Examples for each listing, which demonstrate the use of a particular command or function, are also provided.

**ESCORT Utilities**
contains information on the utility scripts provided on the ESCORT installation diskette.

**Local Screen Generator Utility Program**
describes the operation of the local screen generator utility program provided on the ESCORT installation diskette.

**Appendices**

**Appendix A: Error Messages**
contains an error message directory.

**Appendix B: Debugging Facilities**
provides debugging information.

**Appendix C: AID Subroutines Library**
consists of a program library of the AID subroutines provided on the ESCORT installation diskette.

**Appendix D: Interpretation of Attribute Bytes**
contains information on reading attribute bytes.

**Appendix E: Key Sequences**
consists of tables that present functions and the associated keys and/or key sequences for specific terminal types.

**Appendix F: Environment Variables and Customization**
provides information on setting environment variables and on terminal customization procedures.

**Appendix G: Additional Programs**
contains advanced ESCORT program scripts which can be modified to suit your particular application needs.

**Glossary**
contains definitions for terms and acronyms used in this guide.

**Index**
lists page references for locating specific items in this guide.

# Conventions

## Documentation Conventions

The conventions listed below are used throughout this guide:

- Special function keys on your terminal keyboard are enclosed in a rectangle with rounded corners; for example, ESC.
- Standard alphabetic and numeric keys on your terminal keyboard are printed in bold; for example, **f**.
- Two or more keys separated by spaces indicate that you should press each key sequentially; for example, ESC **f 1**.
- Two keys separated by a hyphen indicate that you should hold down the first key while simultaneously pressing the second key; for example, CTRL – **d**.
- Commands, functions, and keyword operands are printed in bold capital letters. Functions always start with a dollar sign ($). For example, **$SCAN** is a function.
- ESCORT specific key functions and other key functions are printed in capital letters; for example, CLEAR.
- The following type is used to indicate data that the user types at the terminal:

  ***escort script_name***

- The following type is used to indicate information that the system displays on the screen:

  **auto script generation started**

- The following type is used to indicate program text:

  ```
  CONNECT (H1)
  ```

- Multi-word operands are separated by an underscore. For example, *str_expr* represents the words, string expression.
- Brackets [ ] indicate optional operands.
- Braces { } indicate a choice of operands.
- The UNIX file path names are shown with the standard slash character (/). Scripts are portable between the UNIX operating system and the MS-DOS® operating system versions of ESCORT and you may, therefore, substitute the

standard UNIX operating system slash character with the MS-DOS operating system back-slash (\) file name separation character.

**Note**

Throughout this guide, default key combinations are shown for ESCORT specific functions, for example, the ESCORT Interrupt/Resume (I/R) key combination is shown as the key sequence [ESC] **f 2**. These default key combinations may be amended by the System Administrator for your particular environment.

> **If the ESCORT default key combinations are amended you must substitute the amended key combinations for the default combinations shown in the document text, the example scripts and the sample programs in this programmer's guide.**

## Data Entry Conventions

- ESCORT is *case insensitive*, which means that it treats lowercase characters the same as uppercase. The exception to this is string constants.
- The UNIX shell is *case sensitive*, which means that it treats lowercase characters differently from uppercase. Thus, when you invoke ESCORT with parameters from the UNIX shell command line, the parameters, such as file names, must be exactly the same as those in the file system.

## Definitions

The following terms are used throughout this programmer's guide.

**Integer** An integer may include an integer constant, integer variable, integer array element, or an integer function. An integer constant may have any value between $2^{31}-1$ and $-2^{31}+1$, (± 2,147,483,647).

**String** A string may include a string constant, string variable, string array element, or a string function. It may also be a combination of these operands separated by a concatenation operator.

**Host** A host session refers to either a synchronous session or an asynchronous session, unless otherwise specifically noted.

# Other ESCORT Documentation

This Programmer's Guide is part of the *AT&T 3270 Emulator+ ESCORT* documentation. The entire documentation package includes the following:

- **AT&T 3270 Emulator+ ESCORT User's and Programmer's Guides**
  AT&T publication number 308-402.
  This binder contains the following three documents:
  - ESCORT Overview
  - ESCORT User's Guide
  - ESCORT Programmer's Guide
- **ESCORT Quick Reference Card** and Key Sequence Card, AT&T publication number 308-389.

# 2 Programming in ESCORT


**Overview** 2-1

**Program Structure** 2-3

**Character Set** 2-7

**Reserved Words** 2-9

**Constants and Variables** 2-11
Constants 2-11
Variables 2-13

**Operators and Expressions** 2-21
Operators 2-22
Expressions 2-25

**Local Session Screens** 2-29
Local Screen Formats 2-29

**Special Features** 2-35
System Global Variables 2-35
Screen Buffers 2-37
Parameter Passing 2-38

**Synchronous and Asynchronous Host Programming Considerations** 2-41
Synchronous Response/No-Response Mode Transactions 2-41
Asynchronous Communication Port Initialization 2-47
Asynchronous Host Terminal Specification 2-48
Synchronizing Data Transmissions 2-49

# Overview

Read this chapter to learn how to structure an ESCORT program.

The first five sections of this chapter contain the rules and requirements of the language: the program structure, allowable character set, reserved word restrictions, and types of constants, variables, operators, and expressions you may use.

The next section, entitled "Local Session Screens," contains information on how to define and use local screen formats in ESCORT.

The section, "Special Features," contains important notes that you should read before programming in ESCORT. The features covered are: use of screen buffers, the system global variables available, and parameter passing.

The final section in this chapter contains information on how to handle partial synchronous host system responses, the initialization of asynchronous communications ports, and on the method of synchronizing data transmissions for non-screen oriented asynchronous host data applications in ESCORT.

When you have read this chapter you will be able to create programs that automate tasks that previously had to be completed manually and you will be able to design local session screens that provide an interface between you and the host system application.

# Program Structure

The required structure for an ESCORT program is very simple. Each program consists of a series of program statements stored in a file.

## Program Requirements

Every ESCORT program must meet these criteria:

- Contain at least one script. The first script is the main program. Additional scripts are similar to subroutines in other programming languages.
- Begin with a **PROG** (program) statement. You may enter comments before the **PROG** statement if you wish.
- End with an **ENDP** (end of program) statement.

Some rules for structuring ESCORT programs follow:

- Begin each script section with a **SCRIPT** statement and end it with an **ENDS** statement.
- You must declare variables before they can be used in a script.

Optional sections in ESCORT programs include the following:

- You may use local or global variables within your program. The section, "Declaring Global and Local Variables," in this chapter discusses this in more detail. Global variables may be accessed by any script within your program and must be declared in a global variables declaration section following the **PROG** statement. This section is optional and may contain only variable declaration statements such as **INT, CHAR,** and **FIELD.** Local variables are valid only within a particular script and must be declared at the beginning of each script following the **SCRIPT** statement.
- You may define local screen formats. The section, "Local Session Screens," in this chapter provides additional information. Local screen formats are defined in the local

screen format definition section following the **PROG** statement. This section is optional and may contain only local screen definition statements such as **BEGFMT, ENDFMT** and **FIELD.**

## Summary

This diagram outlines the structure of an ESCORT program.

**ESCORT Program Structure**

| | |
|---|---|
| **ABCD** | **PROG S1**<br>**Global Variable Declarations**<br>**INT**<br>**CHAR** **statements only**<br>**FIELD**<br>**Local Screen Format Definitions**<br>**BEGFMT**<br>**FIELD** **statements only**<br>**ENDFMT** |
| **S1** | **SCRIPT**<br>**Local Variable Declarations**<br>**Executable Statements**<br>**ENDS** |
| **S2** | **SCRIPT**<br>**Local Variable Declarations**<br>**Executable Statements**<br>**ENDS** |
| **S3** | **SCRIPT**<br>**Local Variable Declarations**<br>**Executable Statements**<br>**ENDS** |
| | **ENDP** |

- ☐ The program begins with a **PROG** statement followed by an optional global variables declaration section and an optional local screen format definition section.

- The first script, which is required, follows the optional declaration and format definition sections. Additional scripts are optional.
- Each script begins with a **SCRIPT** statement.
- Declaration of local variables follows the **SCRIPT** statement.
- The executable commands and statements of your program follow the local variables declaration. (An exception is the preprocessor command, **COPY**, which may be used anywhere between the **PROG** and **ENDP** statements in your program.)
- Each script ends with an **ENDS** statement.
- The program ends with an **ENDP** statement.

# Character Set

ESCORT allows use of the following characters:

- □ Upper case alphabetic characters (A - Z)
- □ Lower case alphabetic characters (a - z)
- □ Numeric characters (0 - 9)
- □ Special characters (as defined in standard ASCII code).

Some characters or combinations of characters have a specific meaning in ESCORT.

| Character | Use |
|---|---|
| $ | First character in a function name (e.g., $DATE) |
| /* | Starting marker for a comment in a program |
| ( ) | Delimiters for expressions or operand lists |
| − | Minus sign operator |
| + | Plus sign or string concatenation operator |
| * | Multiplication sign operator, or default attribute (FIELD) |
| / | Division sign operator |
| % | Remainder divide (modulo) operator |
| = | Equal sign or assignment |
| != | Not equal sign |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| & | And (logical) |
| \| | Or (logical) |
| ! | Unary Not (e.g., If !((a-b) = 2)...Then) |
| &&n | Preprocessing parameters (n=1 to 9) |
| ? | Comparison operator ($EVAL) |

# Reserved Words

Reserved words have a special meaning in the ESCORT language and should not be used as variable names, program names, labels, or script names.

Listed below are the reserved words:

| | | | |
|---|---|---|---|
| ABEND | DISCON | FRESH | RESET |
| AID | DNLOAD | GETFMT | RETURN |
| ASSIGN(=) | DO | GOTO | RUN |
| ATTN | DUMP | HOME | SCREEN |
| BEEP | DUMMY | IF | SCRIPT |
| BEGFMT | DUP | IGNORE | SERINIT |
| BREAK | EJECT | INS | SHOW |
| BTAB | ELSE | INT | SWITCH |
| CALL | ENDC | LBREAK | SYSAID |
| CAPTURE | ENDFMT | LOG | SYSPRMT |
| CASE | ENDIF | MODE | SYSREQ |
| CHAR | ENDO | NL | SYSRET |
| CHGATTR | ENDP | OPEN | TAB |
| CHKPT | ENDS | PA1 | TEXT |
| CLEAR | ENTER | PA2 | THEN |
| CLOSE | ERASEW | PA3 | TIMEOUT |
| COLOR | ERIN | PF1-24 | TRACE |
| CONNECT | EROF | PGMDMP | UPLOAD |
| COPY | EXIT | PMSG | WAIT |
| CURSOR | FIELD | PRINT | WHILE |
| CYCLE | FM | PROG | WINDOW |
| DEFAULT | FOR | PROMPT | WRITE |
| DEL | FORMAT | READ | WTO |

# Constants and Variables

This section contains information on the types of constants and variables you may use. Any restrictions on their use are also noted in this section.

## Constants

You may use two types of constants in ESCORT. They are

- Integer Constants
- String Constants.

### Integer Constants

An integer constant may have any value between $2^{31}-1$ and $-2^{31}+1$, (± 2,147,483,647), inclusive. A unary minus sign is permitted on an integer constant. Examples of integer constants are

```
 1256
 -126
   13
    0
```

### String Constants

A string constant (or a character string) is a sequence of characters enclosed in double quotation marks. A maximum of 132 characters is permitted in a string constant. Examples of string constants are

```
"HELLO WORLD"
"$10,000.00 / year"
"a"
"I am a string"
```

A string constant may contain upper- and/or lowercase letters.

However, the string constant "HELLO WORLD" and the string constant "hello world" or "Hello World" are not equivalent to each other.

A string can be continued on two or more lines by using the concatenation operator (+). If you use the concatenation operator, you must enclose the expression in parentheses. Below is an example:

```
ADDRESS =    ("26 Bloom Drive,  " +
              "Manchester, N.J. 07728")
```

# Variables

A variable is a symbolic name used to represent a value. This section contains the rules for using variables in an ESCORT program. The value contained in a variable can be changed during execution of a program.

The maximum allowable size for all variables is 64K.

## Naming Variables

- The name of a variable may be up to eight characters long.
- The first character of a name must be alphabetic. The remaining characters may be either alphabetic or numeric.
- An underscore (_) is permitted in a variable name.
- A number sign (#) is permitted in a variable name.
- A field variable name may be prefixed by a format name separated from the simple field name with a period (.).

**Note**
A reserved word may *not* be used as a variable name.

## Declaring Variables

- You must declare your variables at the beginning of your program before they are used in a command statement.
- Integer variables are initialized to zero when you first declare them.
- String variables are initialized to null (no characters) when you first declare them.

### Declaring Global and Local Variables

The scope of a variable may be either **global** or **local**.

- Global Variables

  Declare global variables within the optional declarations section of your program, which starts right after the **PROG** statement and ends at the first **SCRIPT** statement. Once you declare a variable as global, it can be used by any script

within the entire program. A global variable must have a unique name within a program.

- Local Variables

  Declare local variables within the script section of your program. Once a variable is declared it is defined for that particular script only. You may use the same variable name again in another script within the same program.

An example of the use of local and global variables follows:

```
ABCD    PROG      script1
        INT       global1
        CHAR (10) global2
script1 SCRIPT
        INT       local1
        CHAR (10) local2
        .
        .
        ENDS
script2 SCRIPT
        INT       local1
        CHAR (20) local2
        .
        .
        ENDS
        ENDP
```

In the above example, two global variables are declared, *global1* and *global2*. These variables can be used by both *script1* and *script2*. No other variable may be declared using these names.

In *script1*, two local variables are declared, *local1* and *local2*. These two variables are also declared as local variables in *script2*.

## Using Different Variable Types

There are five types of variables used to represent values in ESCORT. They are

- Integer
- Integer Array
- String
- String Array
- Field.

### Integer Variables

An integer variable is a four-byte signed integer that may have a value between $-2^{31}+1$ and $2^{31}-1$, inclusive. Fractional values (decimal numbers) are not allowed; refer to the **$EVAL** function in Chapter 4 for detailed information on how ESCORT handles decimal numbers.

You can declare an integer variable by using an **INT** statement. The **INT** statement is described in detail in Chapter 4.

The value of an integer variable may be changed at any time during program execution.

Listed below are examples of integer declaration statements:

```
INT A           /*Integer A is declared.
INT B           /*Integer B is declared.
INT C           /*Integer C is declared.
INT X           /*Integer X is declared.
 .
 .
 .
A = 20          /*Value of A equals 20.
B = 5           /*Value of B equals 5.
C = (A/4)       /*Value of C equals 5.
X = ((A-B) / C) /*Value of X equals 3.
```

The above example first declares variables *A*, *B*, *C*, and *X* as integers. When you declare the variables they are initialized to zero. Next, the value of each variable is changed by using an assignment statement (=).

### Integer Array Variables

An array is a table of integer variables referenced by the same variable name. An integer array may have a maximum of 2048 elements. These elements can be referred to with subscripts 1 to 2048. Each element in the array must be a four-byte signed integer.

You can declare an integer array by using the **INT** statement. The **INT** statement is described in detail in Chapter 4.

Below is an example of an integer array named *table*. This example shows you how to declare and initialize an array and how to access an array element.

```
INT table (5)               /* table contains 5 elements
                            /* each element has zero value
   .
   .
table = (10,20,30,40,50)     /* each element in the array named table
                            /* is initialized (set to a specific value)
   .
   .
table(2) = -255              /* the 2nd element in the array
                              /* is set to -255
```

The first statement in the example declares an integer array of five elements. Each element in the array *table* is automatically set to zero when it is first declared.

The second statement initializes the array by setting each element to a specific value:

| | | |
|---|---|---|
| 1st array element | = | 10 |
| 2nd array element | = | 20 |
| 3rd array element | = | 30 |
| 4th array element | = | 40 |
| 5th array element | = | 50 |

After execution of the third statement in the example, the value of the second element in the array changes from 20 to −255.

| | | |
|---|---|---|
| 1st array element | = | 10 |
| 2nd array element | = | −255 |
| 3rd array element | = | 30 |
| 4th array element | = | 40 |
| 5th array element | = | 50 |

### String Variables

A string variable may have a maximum of 2048 characters. You specify the maximum length of a variable in the declaration statement.

The value and length of string variables change during program execution depending on the assignment statements in your program. When a string variable is first declared, its length is set to zero and it contains a null string or no data.

You may declare a string variable by using a **CHAR** (character) statement. The **CHAR** statement is described in detail in Chapter 4.

The example below shows how a string variable is declared:

```
CHAR (20) name              /* 20 character string
CHAR (60) address           /* 60 character string
   .
   .
name    =    "Anderson, G.A."
address =    ("26 Bloom Dr.,  " +
             "Manchester, N.J. 07728")
```

In the above example, *name* and *address* are declared as string variables. The *name* has a maximum length of 20 characters. The *address* has a maximum length of 60 characters. When the strings are first declared, they are null strings and therefore have zero length.

In the second part of the example, *name* is assigned a character string constant (Anderson, G.A.) and the current length is therefore set to 14. The *address* is assigned a character string constant (26 Bloom Dr., Manchester, N.J. 07728) and its length is set to 36 (Note that spaces count as characters).

### String Array Variables

A string array is a table of string variables of the same maximum length. Each string array may have a maximum of 2048 elements. These elements can be referred to with subscripts 1 to 2048. Only single dimension arrays may be implemented in ESCORT.

Each element in a string array may contain a character string or a null string. Each element in an array is automatically initialized to null upon declaration.

You can declare a string array by using the **CHAR** statement. The **CHAR** statement is described in detail in Chapter 4.

Below is an example of a string array declaration statement:

```
CHAR (2) tables (5)               /*  array declaration statement
                      .
                      .
tables = ("AB","C","E","GH","IJ")  /*  array initialization
                      .
                      .
tables (2) = "CD"                  /*  assignment of string
                                   /*  "CD" to element 2.
```

The first statement declares a string array. Each element in the array may contain a character string of up to 2 characters. Each element is initialized with zero length.

The second statement uses a special form of the assignment statement for string array initialization to assign specific values to each element of the array. Elements in the array have the following string values and lengths:

| | Value | Length |
|---|---|---|
| tables (1) | "AB" | 2 |
| tables (2) | "C" | 1 |
| tables (3) | "E" | 1 |
| tables (4) | "GH" | 2 |
| tables (5) | "IJ" | 2 |

The last statement in the example sets the second element to "CD." It previously held the value "C."

### Field Variables

The field variable concept is unique to ESCORT. A field variable is a user-defined area in the screen buffer that is associated with a particular screen format. It is also known as a screen field variable.

The **FIELD** statement has two formats. The first **FIELD** format is used to assign a symbolic name (a screen field variable) to a specified area on the screen. This format is used primarily to declare field variables for formatted screens in a host session. See Chapter 4 for further information on this type of **FIELD** statement.

The second **FIELD** format is used to create formatted screens for local sessions and is described more fully in the section, "Local Session Screens", in this chapter.

Both formats use the same naming convention for the field variable:

***[format.]field_name***

Format and field may each be up to eight alphanumeric characters, and the first character must be alphabetic. The format name is optional, but when it is included it must be separated from the field name by a period (.). See the **FIELD, FORMAT, BEGFMT/ENDFMT, GETFMT** and **ASSIGN(=)** statements in Chapter 4 for more detailed information.

Below are examples of field variables:

mainord.orderno
orderno
abc.x1
a.b
racflog.pwd

Screen-field names may be chosen arbitrarily. However, we recommend that you use the actual screen and field names defined for your host application (by, for example, the IMS/VS Message Format Services (MFS) or the CICS/VS Basic Mapping Support (BMS) utility in the synchronous environment). Screen-field names may be used in a string expression or string relational expression. Below are examples:

```
FIELD (5,10,9) mainord.orderno     /* field declaration
CHAR  (3)      prefix              /* character string
    .
    .
prefix = "000"
mainord.orderno = (prefix + "000034")
```

The above statement has the same effect as using

```
CURSOR  (5,10)
TEXT    ("000000034")
```

However, the code written using the **CURSOR** and **TEXT** commands is more difficult to maintain since code updates must be made manually when the host application's screen format changes.

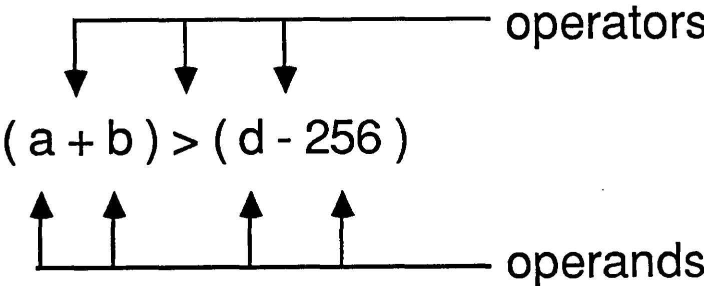
In addition, multiple scripts may need revision since they may all use the same screen. To save time and effort you can define the screen as fields in a separate file which you copy into each script as needed. When an application screen format changes, you have to make only one change to the screen definition file.

# Operators and Expressions

This section contains information on the types of operators and expressions you may use in ESCORT. Definitions of operators, operands, and expressions follow:

- Operators are characters that designate mathematical or relational operations.
- Operands are constants or variables that are acted upon by operators.
- Expressions are combinations of operators and operands. Individual operands may also be considered expressions.

Below is an example of how operators and operands relate to one another:




In this example, *a*, *b*, *d*, and *256* are operands and +, −, and > are operators. The whole statement is an expression as well as its components such as *(a + b)* and *(d − 256)*. In addition, single operands, such as *a*, *b*, *d*, and *256* can be considered expressions.

# Operators

You may use three types of operators in ESCORT. They are

- Arithmetic
- Relational
- String Concatenation.

## Arithmetic Operators

These are characters that designate mathematical operations.

| Operator | Definition |
|---|---|
| + | addition |
| – | subtraction or unary minus (e.g., –1) |
| * | multiplication |
| / | division (The result is always truncated (e.g., 36/5 = 7.) |
| % | remainder division (modulo). For example, 36 % 5 = 1. The remainder of 36 divided by 5 equals 1. |

## Relational Operators

These are characters that compare two values and yield either a true (non-zero) value or a false (zero) value.

<table>
<tr><th>Operator</th><th>Definition</th></tr>
<tr><td>=</td><td>equal</td></tr>
<tr><td>!=</td><td>not equal</td></tr>
<tr><td><</td><td>less than</td></tr>
<tr><td>></td><td>greater than</td></tr>
<tr><td><=</td><td>less than or equal</td></tr>
<tr><td>>=</td><td>greater than or equal</td></tr>
<tr><td>&</td><td>logical AND (IF (a>b) & (c=d) THEN)</td></tr>
<tr><td>|</td><td>logical OR (WHILE (f1="xyz") | (y=230) DO)</td></tr>
<tr><td>!</td><td>unary NOT (WHILE !((f1="xyz") | (y=230)) DO)</td></tr>
</table>

The operands compared by relational operators must be both integers or both strings. You may not compare integers to strings.

Comparison of strings is based on ASCII collating sequence order. For example, the following operations will return a true (non-zero) value:

```
"AA"                 !=   "AB"
"I AM A STRING"       =   "I AM A STRING"
"Kg"                  >   "KG"
```

Comparison of integers is based on the values of the two expressions compared. For example, the following operation will return true (non-zero) since the value of the first expression is less than the value of the second expression.

```
(4 + 2) < (5 * 2)
```

Use the **$EVAL** function to compare floating point values. The **$EVAL** function is described in detail in Chapter 4.

### String Concatenation Operators

You may use a concatenation operator to chain (or link) together string operands. A string operand can be a string constant, string variable, string array element, screen-field variable, or string function.

Either a plus sign (+) or a space can be used as a concatenation symbol. If you use a concatenation operator, you must enclose the expression in parentheses.

These two statements are equivalent:

```
PRINT ($DATE + " " + $TIME + " TEST0024 ENDED")
PRINT ($DATE " " $TIME " TEST0024 ENDED")
```

The first statement uses a plus sign (+) as a concatenation operator and the second statement uses a space. Additional spaces between operands are ignored.

## Precedence of Operators

Operators in an expression are evaluated in the order shown in the table below. Operators on the same line of the table have the same precedence; rows are in order of decreasing precedence. Operators with the same precedence are evaluated from left to right as they appear in an expression.

| Operator | Definition |
| --- | --- |
| ( ) | parentheses |
| unary – ! | unary minus, unary NOT |
| * / | multiplication, division |
| % | remainder division (modulo) |
| + – | addition, subtraction |
| = != < > <= >= | equal, not equal, less than, greater than, less than or equal, greater than or equal |
| & | logical AND |
| \| | logical OR |

# Expressions

You may use three types of expressions in ESCORT. They are

- Integer
- Relational
- String.

## Integer Expressions

An integer expression consists of a single integer operand or multiple operands separated by arithmetic operators. Expressions containing multiple operands must be enclosed in parentheses. An integer operand may be an integer constant, integer variable, integer array element, or integer function.

An integer expression that has multiple operands is known as a compound expression. Compound expressions must be enclosed in parentheses.

Below are examples of integer expressions:

| Expression | Definition |
|---|---|
| 256 | integer constant |
| −1 | integer constant |
| a | integer variable |
| i | integer variable |
| c(2) | integer array element |
| (a+(b-256) * 8) | compound expression |
| ($GETCUR+1) | compound expression using an integer function |

The result of an integer expression is a signed integer.

Refer to the **$EVAL** function in Chapter 4 for detailed information on how ESCORT handles decimal numbers.

## Relational Expressions

A relational expression is any expression containing a relational operator (=, !=, <, >, <=, >=, &, |, !). The operands in the expression can be either string type or integer type. The result of a relational expression is true (non-zero) or false (zero).

Relational expressions are usually used in the clause of an **IF** or **WHILE** statement. However, they can also be assigned to an integer variable or an integer array element.

Below are examples of relational expressions:

- Two integer expressions separated by a relational operator. In this example, *a*, *b*, *c*, and *d* are integer variables.

```
IF  a=2  THEN ...
WHILE (a+2) != c DO ...
IF  b<=d THEN ...
```

- Two strings separated by a relational operator. In this example, *f1*, *f2*, and *f3* are character or field variables.

```
IF f1="xyz" THEN ...
IF f2=f3  THEN ...
```

- Two or more simple relational expressions separated by & or | operators. In this example, *a* and *b* are integer variables and *f1* and *f2* are string variables.

```
IF  ((a=b) | (f1="2300")) & (f2 > f1) THEN ...
```

### String Expressions

A string expression consists of a single string operand or multiple string operands separated by a string concatenation operator (a plus sign or blank space). A string expression containing multiple string operands must be enclosed in parentheses. A string operand can be a string constant, string variable, string array element, screen field variable, or string function.

Below are examples of string expressions:

| Expression | Definition |
|---|---|
| "a" | single character |
| "I am a string" | one string |
| ("HELLO"+"WORLD") | two strings concatenated |
| ("time is " + $TIME) | string and a string function |
| (A  B  "string") | two variables and a string constant |

The result of a string expression is a character string.

In the next example, the variable *name* contains the character string *AT&T - Information Systems*.

```
CHAR (30) name
CHAR (20) division
     .
     .
division = "Information Systems"
name   = ("AT&T - " DIVISION)
```

# Local Session Screens

## Local Screen Formats

The local session feature of ESCORT allows the user to develop a UNIX operating system-based, front-end to a host application, known as a Local Screen Format.

The local session screen feature of ESCORT allows you to create local session screens tailored to specific user's needs. Local screens can be created using various attributes that provide enhanced characteristics similar to host application screen formats.

### Local Screen Format Definition Area

Local screen formats are defined in the local screen format definition section of a script. The local screen format definition section is optional, but if local screen formats are to be defined the definition section starts after the **PROG** statement and ends at the first **SCRIPT** statement.

It is good programming practice to define local screen formats immediately after declaring global variables.

The **BEGFMT** statement indicates the beginning, and the **ENDFMT** statement indicates the end of local screen format definition areas for each screen format name.

ESCORT allows for up to 100 local screen formats in a single script, each screen format name must, therefore, be unique within a script.

Each local screen format may have up to 500 **FIELD**s.

An example of the use of **BEGFMT** and **ENDFMT** follows:

```
progl     PROG     main
          .
          .
          BEGFMT order_1
            (Field statements)
          ENDFMT

          BEGFMT order_2
            (Field statements)
          ENDFMT

main      SCRIPT
          .
          .
```

In the above example, two local screen formats, named *order_1* and *order_2*, are defined.

Local screen formats can be defined either locally within a script, or externally in another file. A **COPY** command can be used to include externally defined local screen formats in the script at run time. See the **COPY** command in Chapter 4 for additional information.

You may create an unformatted screen by using a **BEGFMT** and an **ENDFMT** statement without an intervening **FIELD** statement. An unformatted screen does not contain any attribute characters and therefore appears as one unprotected field of 1920 characters.

## Multiple Format Files

Of the 100 local screen formats that you can define in ESCORT, six formats are retained in memory. The remaining formats are *spilled* and are written to individual format files. *Spilled* formats are loaded into memory upon demand.

ESCORT creates one file for each *spilled* format in your current directory. The file name of each *spill* file is

*screen_name*.$fm

where *screen_name* is the screen name defined by each **BEGFMT** statement.

You should delete *spilled* local screen format files from your current directory following execution of your script.

Do not delete *spilled* local screen format files that are produced when a script is parsed and an executable run-time script is

created. You must ensure that the appropriate *spilled* local screen format files are available in your current directory when interpreting (executing) a run-time script. Execution of a run-time script will fail if the expected *spilled* files are not found. These *spilled* files may be deleted after the run-time script has been interpreted (executed).

## Defining Local Screen Formats

Individual fields within a local screen format are defined by the **FIELD** statement.

You can design a local screen format that contains almost all of the characteristics of the actual host application screen. The **FIELD** statement used for local screen formatting has an attribute list that allows for the definition of Primary Attributes and Extended Field Attributes, similar to the IBM® 3270 screen formats.

The Primary Attributes allow you to define data fields as protected, unprotected, numeric, alphabetic, highlighted, non-displayable, or with a pre-modified data tag. The Extended Field Attributes provide enhancement to the field by defining such characteristics as blinking, reverse video and underlining. See the **FIELD** statement in Chapter 4 for a complete list of definable attributes.

See Chapter 6 for information on using the Local Screen Generator Utility Program provided on the ESCORT installation diskette.

To create a local formatted screen, the **FIELD** statement must be defined within a local screen format definition area that starts with a **BEGFMT** and ends with an **ENDFMT** statement.

You should define local screen format fields carefully since ESCORT will *not* check for overlapping field definitions. Results may be unpredictable if data fields overlap. Areas on screen that are not defined by a **FIELD** statement are automatically treated by ESCORT as protected, numeric fields.

A field variable is used to name the screen area defined by the **FIELD** statement. See the section "Field Variables" in this chapter for further information on naming field variables.

You may also use the **FIELD** statement to define literal fields to make your local application screen more readable. Literal fields

are defined using the keyword *DUMMY* in place of the field variable. The literal field narrative is established by adding an argument to the *DUMMY* keyword. Literal fields should be created as protected fields to prevent users from overwriting the literal field narrative.

An example of the use of the **FIELD** statement follows:

```
progl   PROG    main
        .
        .
        BEGFMT logon
          FIELD (10,12,9,(P,*,H,*,*,*,*)) DUMMY "PASSWORD:"
          FIELD (10,22,8,(*,*,D,*,*,*,*)) passwd
        ENDFMT
main    SCRIPT
        .
        .
```

In the above example, a local screen format, *logon*, is declared. A literal field that is Protected and Highlighted, and contains the prompt narrative, *PASSWORD:* is followed by the screen variable, *passwd*. The *passwd* screen variable has a non-displayable (dark) field attribute which means that, when entered by the operator, the characters typed will not be echoed back to the terminal screen.

## Loading Local Screen Formats

A local screen format is loaded into the screen buffer in memory by use of the **GETFMT** command in an ESCORT script.

The **GETFMT** command loads the specified local screen format into the associated local session's presentation space. Only one local screen format can be loaded within a local session's presentation space at any given time. However, you can change the format in a presentation space by executing another **GETFMT** command.

ESCORT allows you to load the same local screen format into more than one local session's presentation space. See the **GETFMT** command in Chapter 4 for further information.

In the following example the local screen format, *logon*, is loaded into local session L1 and the script enters Interactive mode to allow the user to enter the required data.

```
progl   PROG    main
        .
        .
        BEGFMT logon
          FIELD (10,12,9,(P,*,H,*,*,*,*)) DUMMY "PASSWORD:"
          FIELD (10,22,8,(*,*,D,*,*,*,*)) passwd
        ENDFMT
main    SCRIPT
        .
        .
        GETFMT (L1, logon)
        EXIT
        .
        .
```

# Special Features

## System Global Variables

The following system global variables are available to users of ESCORT.

### SCREEN

The value of *SCREEN* is a 1920-character string that contains the screen image. ESCORT converts nulls, attributes, and nondisplayable characters to blanks while copying the current screen image to the specific screen buffer. For more details, see the following section, "Screen Buffers".

### SYSAID

The value of *SYSAID* is an integer. It contains the code for the last AID key pressed by the operator while in Tutorial (or Interactive) mode. See the **EXIT** command for more information about using *SYSAID*.

### SYSPRMT

*SYSPRMT* is a string variable. It contains the asynchronous host system prompt, and optionally, the screen column and row position for the first character in the prompt string. The *SYSPRMT* variable is initialized by the **PROMPT** command. See the **PROMPT** and **WAIT** commands in Chapter 4 for information on initializing and using the *SYSPRMT* variable.

### SYSRET

The value of *SYSRET* is an integer. It contains the return code after

- **OPEN, CLOSE, READ, WRITE**, and **CHKPT** file operations
- the asynchronous environment **WAIT** command
- the **CAPTURE ON, CONNECT, DISCON, LOG, RUN** and **PUTENV** commands
- the Interactive or Tutorial mode **TIMEOUT** command
- the **DUMP** debugging command.

See the appropriate commands in Chapter 4 for more information about using *SYSRET*.

## Screen Buffers

ESCORT maintains an image of the last refreshed screen for each host and local session in separate presentation spaces or screen buffers.

ESCORT is able to manipulate data in the screen buffer of the currently connected host or local session. Data can be moved between the presentation spaces of separate sessions by use of the **CONNECT** command and **ASSIGNMENT** statement. ESCORT can perform the following functions on the data contained in the connected session screen buffer:

- Retrieve data from the buffer.
- Write data into the buffer.
- Compare a screen field to a literal.
- Search the buffer for a particular character string.
- Get a substring from a screen field.
- Find out the length of a screen field.
- Find out the current cursor position.
- Log the screen image to a file.
- Print the screen image.
- Examine field attributes.

In addition to the features listed above, you can use all standard terminal key functions with the connected session screen buffer.

A special, internally declared string variable (or system global variable), *SCREEN*, is available to you to access the connected session screen buffer. This variable may be used in the same way as any other string except that it may not be the target of an assignment.

For further information on screen buffers, see the **SHOW** and **CONNECT** commands in Chapter 4.

## Parameter Passing

You may pass up to nine parameters by specifying the parameters on the command line when executing a non-run-time ESCORT script.

For example,

```
escort script IMSCMD,"LOGON.S",,5,X_YZ
```

The preprocessing parameters are named &&1, &&2, &&3, &&4, &&5, &&6, &&7, &&8, and &&9. The string "&&" is reserved by ESCORT to identify preprocessing variables.

The values assigned by the above example are:

| Parameter | Value |
|---|---|
| &&1 | IMSCMD |
| &&2 | "LOGON.S" |
| &&3 | null string |
| &&4 | 5 |
| &&5 | X_YZ |
| &&6 | null string |
| &&7 | null string |
| &&8 | null string |
| &&9 | null string |

Each parameter may contain a maximum of 100 characters. No blanks may be used within parameters or to separate parameters. You may use an underscore ( _ ) character within a parameter as a blank (space character).

The parameters passed on the command line are substituted by ESCORT before it performs syntax checking on each command. The values for these parameters are determined strictly by position on the command line. Refer to the section, "Command Line" in Chapter 2 in the *ESCORT User's Guide* for information on passing parameters to a run-time script.

This program code relates to the above example and shows you how to use parameter passing in ESCORT.

```
COPY   &&2           /* Copies "LOGON.S"
TEXT   ("/for &&l")
FORMAT &&l
x   =  .fieldl
y   =  .field2
j   =  &&4           /* j = 5
a   =  "&&5"         /* a = "X YZ"
```

# Synchronous and Asynchronous Host Programming Considerations

## Synchronous Response/No-Response Mode Transactions

No-response mode transactions permit multiple transmissions from a synchronous host before returning a full response. When the active synchronous host system receives such a transaction, it may send a keyboard unlock command to the originating terminal.

This unlock response poses a special problem for ESCORT applications since proper execution of a script depends on getting the full transaction response.

Ten scripts, known as AID subroutines, are available on your installation diskette. Each one is a complete ESCORT script and can be used to deal with the problem of the early unlock sent by the synchronous host.

By using the AID subroutines, you are able to specify a set of parameters that define a particular condition. Each subroutine executes a specified AID key and then monitors the screen for the defined condition. Control is returned to the calling script only when the condition has been satisfied.

For example, the AID subroutine, aid_cc

- □ moves the cursor to the last position on the screen (row 24, column 80),
- □ sends the specified AID key,
- □ waits for the cursor position on the screen to change,
- □ returns control to the calling script when the cursor position changes.

You can add the necessary subroutines to your program by using the preprocessor command, **COPY**.

For example, to copy the AID subroutine, aid_cc, use the following code:

```
COPY "/usr/escort/slib/aid_cc"
```

This statement copies aid_cc from the subroutine library in the directory named */usr/escort/slib*.

A copy of the complete text of each AID subroutine script is available in Appendix C.

The special key sequence, ESC **f 0**, activates or deactivates AID subroutine substitution while in Automatic Script Generation in Interactive mode, when connected to an active synchronous host session. Each time an AID key is encountered in the automatically generated script, ESCORT generates a subroutine call to the script named *aid_resp*. Refer to the section, "Automatic Script Generation" in Chapter 2 in the *ESCORT User's Guide* for further information on the use of ASG.

## AID Subroutines

Following is a list of the ten AID subroutines that are available on your installation diskette, along with the proper format for invoking each subroutine in your program. The AID key codes are listed at the end of this section.

**aid_gc** — **Wait for tag to disappear.**

CALL aid_gc (key_code)

Writes a tag character at the next to last position on the screen. Sends an AID key and waits until the tag has disappeared. In order for this routine to work properly, screen position 1919 (row 24, column 79) must be unprotected.

**aid_cc** — **Wait for cursor position to change.**

CALL aid_cc (key_code)

Moves cursor to the last position on the screen. Sends an AID key and waits until the cursor is no longer in that position.

**aid_01c** — **Wait for line 1 to change.**

CALL aid_01c (key_code)

Sends an AID key and waits until any character on line 1 has changed.

**aid_24c** — **Wait for line 24 to change.**

CALL aid_24c (key_code)

Sends an AID key and waits until any character on line 24 has changed.

**aid__lc** **Wait for specified line to change.**

CALL aid__lc (key__code, row)

Sends an AID key and waits until the specified line has changed. The *row* is the line in which the contents must change when the full response arrives from the synchronous host. It can be an integer constant or an integer variable.

**aid__fc** **Wait for field to change.**

CALL aid__fc (key__code, field__name)

Sends an AID key and waits until the specified field has changed. The *field__name* is the name of the field in which the contents must change when the full response arrives from the synchronous host. It can be a screen-*field__name* or a short name.

**aid__sma** **Wait for specified message to appear.**

CALL aid__sma (key__code, msg, row, col, length)

Sends an AID key and waits until a specified message has arrived in the screen buffer. The *msg* is the expected message and can be either a string constant or a string variable. The *row* and *col* specify the row and column address where the search begins. The *length* specifies the number of characters.

**aid__smd** **Wait for specified message to disappear.**

CALL aid__smd (key__code, msg, row, col, length)

Sends an AID key and waits until the specified message has disappeared from the screen. The *msg* is the expected message and can be either a string constant or a string variable. The *row* and *col* specify the row and column address where the search begins. The *length* specifies the number of characters.

**aid_kc** **Wait for tag field to be overwritten by synchronous host system response.**

CALL aid_kc (key_code)

Writes a PF key in row 24, column 74, sends an AID key, and waits until the tag has been overwritten by a response from the synchronous host system. In order for this routine to work properly, the five characters starting at screen position 1914 (row 24, column 74) must be unprotected.

**aid_resp** **Wait for cursor position to change (used in Automatic Script Generation.)**

CALL aid_resp (key_code)

This is a generic subroutine which may be modified to suit your particular application environment. Currently, this subroutine moves the cursor to the last position on the screen, sends the AID key, and waits until the cursor has moved to another location on the screen. This subroutine is used when you press (ESC) **f 0** to activate or deactivate AID subroutine substitution while in Automatic Script Generation (ASG) mode.

## AID Key Codes

The key codes representing the AID keys are:

| AID key | Code |
|---|---|
| ENTER | 0 |
| PF1 | 1 |
| PF2 | 2 |
| PF3 | 3 |
| . | . |
| . | . |
| PF23 | 23 |
| PF24 | 24 |
| CLEAR | 25 |
| PA1 | 26 |
| PA2 | 27 |
| PA3 | 28 |
| ATTN | 29 |
| SYS_REQ | 30 |

## Asynchronous Communication Port Initialization

The ESCORT statement, **SERINIT**, is used to define all of the parameters necessary for establishing the line connection to an asynchronous host. These parameters must be provided before the asynchronous host session is physically connected by a **CONNECT** command.

The system global variable, *SYSRET*, returns the result of a **CONNECT** command. A failed **CONNECT** (*SYSRET* value of −1) may indicate one of several error conditions: either the communication port parameters have not first been provided using a **SERINIT** statement, or one or more of the initialization parameters is incorrect. To assist you in correcting the initialization parameters, ESCORT writes various error messages to the file named *escort.pr{proc-id}* created in the directory defined by the ESCDIR environment variable.

If you specify an asynchronous session as the *session-id* parameter to a **PROG** command, the ESCORT script is initially connected to the associated screen buffer only, since a **CONNECT** command, preceded by a **SERINIT** statement, is required to make the physical connection.

The first **CONNECT** command, to a particular asynchronous host, in a script makes the connection to the host using the parameters provided by the preceding **SERINIT** statement. The connection is not dropped when, for example, a connection to another host system is made, (logoff procedures and a **DISCON** command are used if the connected session *is* to be dropped). Subsequent connections to the asynchronous host reactivate the existing connection.

If new parameters are provided by a second or subsequent **SERINIT** statement, a succeeding **CONNECT** command establishes a new connection using the second set of parameters.

Refer to the **CONNECT** and **SERINIT** commands in Chapter 4 for information on the command format and for an example.

## Asynchronous Host Terminal Specification

Some asynchronous applications request terminal type information. You should specify your terminal as a DEC® VT100™ on these remote asynchronous hosts, regardless of the actual terminal type being used.

## Synchronizing Data Transmissions

ESCORT provides you with the ability to scan the data received from an asynchronous host in order to synchronize the sending of data and commands from a script.

### Scanning Asynchronous Host Data

The synchronization of data transmission problem is similar to response/no-response mode transactions in the synchronous environment. Proper script execution depends on receiving an entire transaction response from the asynchronous host. However, unlike the synchronous host system response where a complete screen can be scanned for the anticipated string, data from an asynchronous host is transmitted in a stream; that is, it is not screen oriented, and the exact location of a particular transaction response may not be known.

Two ESCORT commands, **PROMPT** and **WAIT**, are available to assist with the scanning of a stream of asynchronous host data.

The special asynchronous version of the **WAIT** command provides for up to eight search string parameters, control is returned from the command when one of the search string parameters is detected in the incoming data stream. If none of the search string parameters is detected within the **WAIT** command timeout period, control is automatically returned to the script. ESCORT assigns the positional number of the search string detected in the asynchronous host response, to the global system variable, *SYSRET*. A value of −1 is returned in *SYSRET* if no parameter is detected.

### Asynchronous Host System Prompts

A specialized parameter is also available to assist in detecting asynchronous host system prompts. The **PROMPT** command is used to initialize the system global variable, *SYSPRMT*. The parameters to the **PROMPT** command allow you to define the asynchronous host system prompt, for example, the UNIX operating system default dollar sign ($) prompt, and to define the column only, or column and row, screen position of the prompt.

ESCORT automatically assigns the **PROMPT** command parameters to the system global variable, *SYSPRMT* which, in turn, is used as a search string parameter in the **WAIT**

command. Use of this special parameter, *SYSPRMT*, in your script provides increased flexibility; if, for example, the asynchronous host system prompt is altered, you need only change the parameter in a single **PROMPT** command to effectively amend all necessary **WAIT** commands in your script.

Refer to the **PROMPT** and **WAIT** commands in Chapter 4 for detailed information on command format.

## Automatic Script Generation

ESCORT automatically includes suitable **PROMPT** commands that specify a dollar sign ($) in screen column 1 as the asynchronous host response in scripts generated using Automatic Script Generation in an asynchronous environment. In addition, the generated script includes a **WAIT** command referencing the system prompt, following every **TEXT** statement. Refer to the section, "Automatic Script Generation" in Chapter 2 in the *ESCORT User's Guide* for further information on the use of ASG.

# 3 Sample Programs


**Overview** 3-1

**Synchronous Host Sample Program** 3-3
Program Execution 3-5
Program and File Listings 3-24

**Asynchronous Host Sample Program** 3-39
Program Execution 3-41
Program and File Listings 3-60

# Overview

This chapter contains the program listings for two complete sample programs. Sections of program listings are discussed, local session and host screens are shown, and important functions, such as error handling, are reviewed.

The programs demonstrate how ESCORT works in both the synchronous and asynchronous environments, provide examples of program structures, and show how ESCORT programs are executed.

Complete listings of all scripts and files are given at the end of both sections. Logical sections of program are also presented in the chapter, with explanations of their operation and sample screens.

These samples are provided to demonstrate how ESCORT works and are dependent on specific host applications. For this reason the programs are not included on your ESCORT installation diskette.

Refer to Appendix G for information on additional ESCORT scripts that you may be able to modify for your particular application.

# Synchronous Host Sample Program

This sample program provides a new front-end for users who are responsible for adding customer information to a synchronous host data base. The program performs the login procedure, prompts for customer information, takes corrective action if the user enters an invalid zip code, and logs the user off when necessary.

This sample program is similar to the asynchronous host sample program provided in this chapter. Compare the two samples to review the differences in the code necessary in the two environments.

The sample comprises two scripts, a main program, *addcust.p* and a subroutine, *loginims.s*; together with two local screen format files, *addcust.l* and *login.l*; and two host screen format files, *custadd.f* and *chkzip.f*.

The login subroutine, *loginims.s*, accepts login information from the user via a local session screen and automates the synchronous host login procedure. The main program, *addcust.p*, takes customer information, entered by the user in a local session screen, and updates a synchronous host data base.

Execution of the program is subdivided into six main sections:

- Declaration of variables and definition of screen formats.
- Log in to synchronous host application.
- Collect new customer information.
- Update synchronous host data base with new customer information.
- Log off from host.
- Copy subroutines.

Two error routines are demonstrated:

- Failure to log in to host.
- Zip code entered does not match customer's city and state.

# Program Execution

## Declarations and Definitions

The first section of the *addcust.p* program comprises the Global Variable Declarations section and the Local and Host Screen Format Definition sections.

### Global Variable Declarations

The three subsections in the Global Variables declaration section declare variables for use with the synchronous host login and customer information procedures, and for general use.

Five host login variables are declared, each 8 characters in length:

```
char(8) applic            /* host application id
char(8) racfid            /* RACF User id
char(8) racfpwd           /* RACF User password
char(8) cssid             /* application User id
char(8) csspwd            /* application User password
```

Ten customer information variables are declared with the character lengths indicated:

```
char(8)    branch
char(40)   name
char(30)   street
char(9)    geocd
char(15)   city
char(2)    state
char(5)    zip
char(5)    areacd
char(3)    nnx
char(4)    exch
```

Two miscellaneous variables are declared:

```
int        fldpos
char(1)    tryagain
```

### Local Screen Format Definitions

The two local screen formats, *addcust.l* and *login.l*, are defined in this section:

```
copy  "addcust.l"        /* customer information screen
copy  "login.l"          /* login parameters screen
```

The COPY preprocessor command inserts the content of the *addcust.l* and *login.l* local screen format files into the main program.

Both local screen format files use the *DUMMY* keyword and a literal to produce field narratives on the screen. The attribute

lists define certain fields as Protected or Unprotected, and reverse video or normal display. Refer to the **FIELD** command in Chapter 4 for a complete list of definable attributes.

The *addcust.l* local screen format has the defined screen name, addcust:

```
begfmt addcust
       field (1,30,20,(P,A,H,R,R,7,0)) DUMMY "CUSTOMER ADD SCREEN"
       field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "Service Branch: "
       field (3,22,8, (U,A,H,R,R,7,0)) addcust.branch
       field (5,5,9,  (P,A,H,R,N,7,0)) DUMMY "Name:    "
       field (5,15,35,(U,A,H,R,R,7,0)) addcust.name
       field (7,5,9,  (P,A,H,R,N,7,0)) DUMMY "Street:  "
       field (7,15,35,(U,A,H,R,R,7,0)) addcust.street
       field (9,5,7,  (P,A,H,R,N,7,0)) DUMMY "City: "
       field (9,15,15,(U,A,H,R,R,7,0)) addcust.city
       field (9,32,7, (P,A,H,R,N,7,0)) DUMMY "State: "
       field (9,40,2, (U,A,H,R,R,7,0)) addcust.state
       field (9,44,10,(P,A,H,R,N,7,0)) DUMMY "Zip Code: "
       field (9,55,5, (U,A,H,R,R,7,0)) addcust.zip
       field (11,5,12,(P,A,H,R,N,7,0)) DUMMY "Phone No.: "
       field (11,18,3,(U,A,H,R,R,7,0)) addcust.areacd
       field (11,23,3,(U,A,H,R,R,7,0)) addcust.nnx
       field (11,28,4,(U,A,H,R,R,7,0)) addcust.exch
       field (15,5,21,(P,A,H,R,N,7,0)) DUMMY " Press PF12 to EXIT. "
       field (24,2,70,(P,A,H,R,N,7,0)) addcust.status
endfmt
```

The *login.l* local screen format has the defined screen name, login. Note that this local screen format makes use of the non-displayable (dark) attribute for the two password fields:

```
begfmt login
       field (1,28,23,(P,A,H,R,R,7,0)) DUMMY "HOST LOGIN SCREEN"
       field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "Application Id: "
       field (3,22,8, (U,A,H,R,R,7,0)) login.applic
       field (5,5,16, (P,A,H,R,N,7,0)) DUMMY "RACF User Id:   "
       field (5,22,8, (U,A,H,R,R,7,0)) login.racfid
       field (7,5,16, (P,A,H,R,N,7,0)) DUMMY "RACF Password:  "
       field (7,22,8, (U,A,D,R,N,7,0)) login.racfpwd
       field (9,5,16, (P,A,H,R,N,7,0)) DUMMY "CSS User Id:    "
       field (9,22,8, (U,A,H,R,R,7,0)) login.cssid
       field (11,5,16,(P,A,H,R,N,7,0)) DUMMY "CSS Password:   "
       field (11,22,8,(U,A,D,R,N,7,0)) login.csspwd
       field (24,2,70,(P,A,H,R,N,7,0)) login.status
endfmt
```

## Host Screen Format Definitions

Two host screen formats are also inserted into the main program by the following **COPY** commands:

```
copy   "custadd.f"          /* customer administration screen
copy   "chkzip.f"           /* zip code screen
```

The listings for these two files are shown at the end of this section.

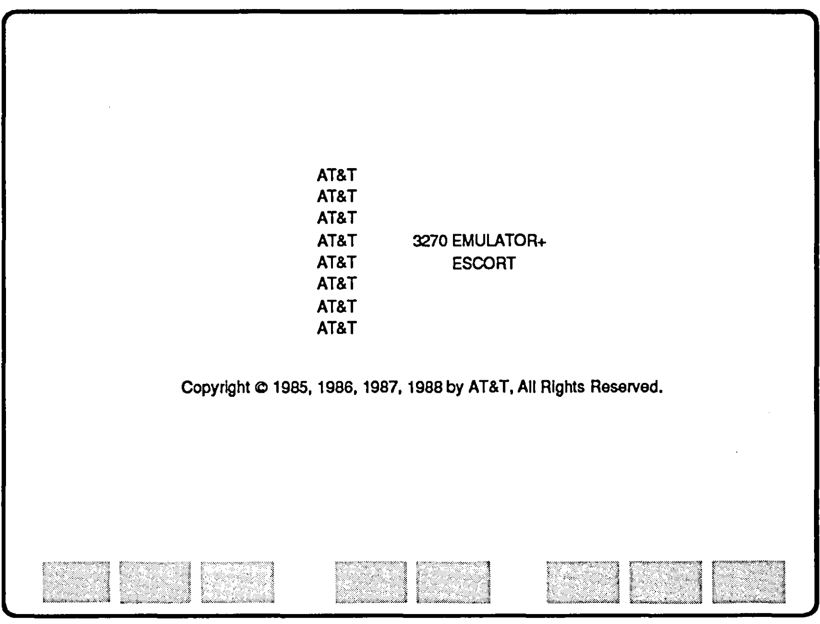
# Log in to Synchronous Host

## Execute Main Program

To run this sample program, at the UNIX shell prompt, the user types

***escort addcust.p***

and presses RETURN. The following ESCORT banner screen is displayed briefly




The first program line

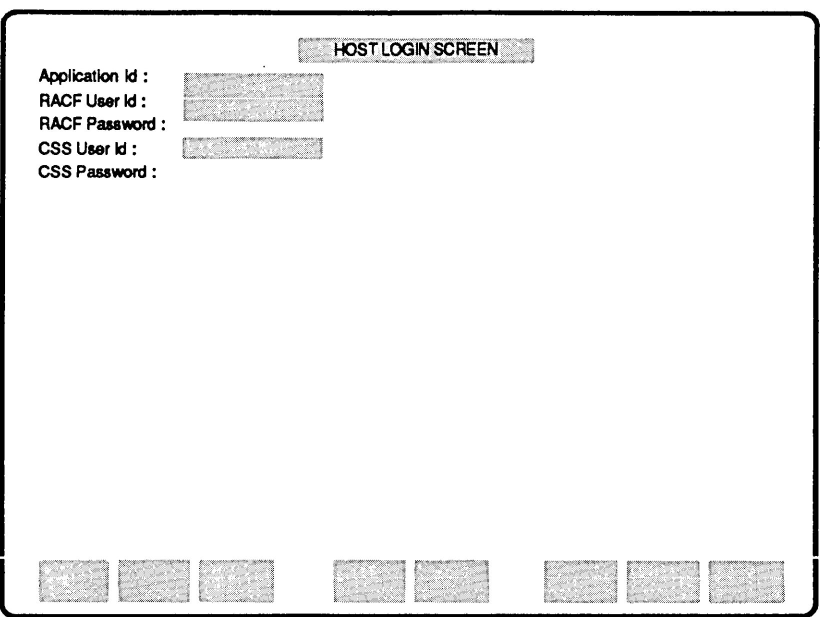
```
add_cust  prog  main (L1)
```

indicates the beginning of the program, identifies the first script named *main* and connects to local session, L1. A local variable is declared in the first script:

```
main      script
          int  rtncode                /* subroutine return code
```

## Load Formats and Enter Data

The next section of program loads the two local screen formats, *login.l* and *addcust.l* into local sessions, L1 and L2 respectively. The *login.l* screen is activated and displayed:




The program exits to Tutorial mode to allow the user to enter the appropriate login data:

```
getfmt (L1, login)          /* assoc local scrn fmt with L1
getfmt (L2, addcust)        /* assoc local scrn fmt with L2
rtncode = 1
while (rtncode != 0)         /* while log in failed
  do
    show (L1)               /* display local session 1
    exit                    /* exit to tutorial mode
```

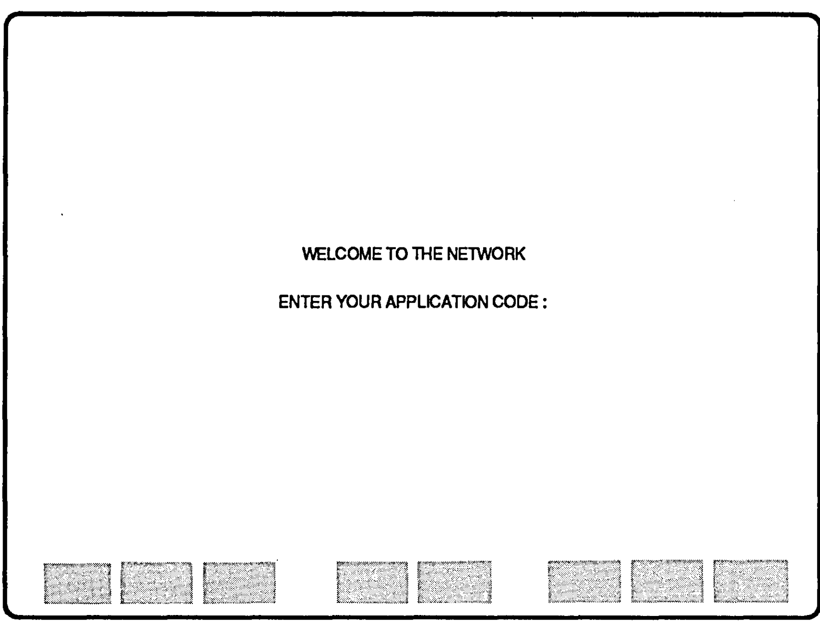
### Assign Data

When program execution is resumed, after the login data is entered and (RETURN) is pressed, the program saves the data entered via the *login* local screen format by assigning the data to global variables for later use. The **FORMAT** command defines the default format as *login*; it is not necessary, therefore, to assign the format name to individual fields.

```
format login
applic = .applic        /* host application id
racfid = .racfid
racfpwd = .racfpwd
cssid = .cssid
csspwd = .csspwd
```

### Activate Synchronous Host Session

The main program activates synchronous host session H1. The value of the system global variable, *SYSRET*, is checked to determine whether the connection to the host is successful. If the connection failed, the attributes for the *status* field, initially defined as Protected to prevent users from writing to this area, are changed to Unprotected to allow the program to write the *Host System Not Available* error message to the *status* field. The *status* field attributes are changed back to Protected after the error message is written. If the connection is successful, the program waits for the following synchronous host session sign-on screen to appear:




```
connect (H1)            /* activate host session 1
if (sysret = -1)
then
  connect (L1)
  fldpos = $fldaddr(login.status)
  chgattr (L1, fldpos, (U,*,H,*,R,*,*))
  login.status = ("Host System Not Available.")
  chgattr (L1, fldpos, (P,*,H,*,R,*,*))
  rtncode = 2
  cycle
endif
show (H1)               /* display host session 1
while !($scan("WELCOME")) /* wait for sign-on screen
  do
    fresh
  endo
```

The *addcust.p* program calls the *loginims* subroutine and passes six parameters:

```
call loginims(applic,
              racfid,
              racfpwd,
              cssid,
              csspwd,
              rtncode)
```

### Log in Procedure

Review the program listing for the *loginims.s* subroutine at the end of this section. The **CALL** command invokes the *loginims* subroutine to log in to the IMS application. The *loginims* **SCRIPT** statement has a declaration list corresponding to the parameter list in the *addcust* **CALL** command.

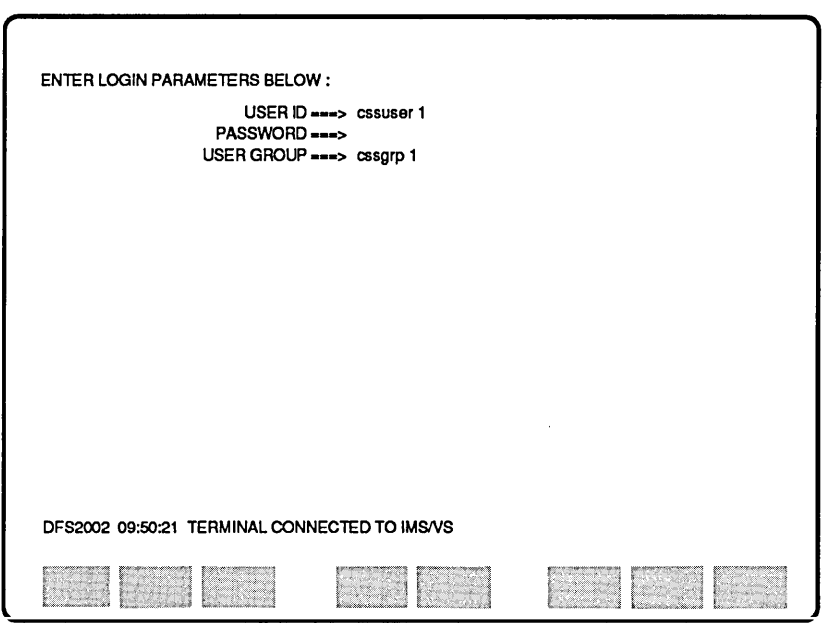
The *loginims* subroutine returns one of three codes to the *addcust* program via the *rtncode* variable:

0 = successful login
1 = login rejected
2 = system not available.

The *loginims* subroutine

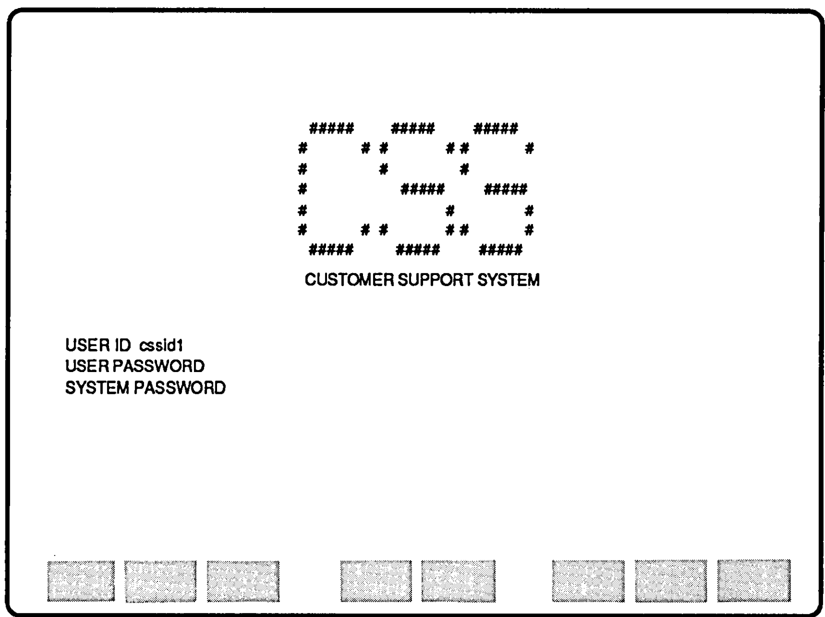
1. Enters the application code.
2. Waits for the sign-on screen.
3. Checks for system failure. If the synchronous host system is not available, the return code is set to 2, an image of the screen is logged, a message is issued, and the *login* local screen is redisplayed.

4 Enters login information. The following screen shows the login data automatically entered by the ESCORT subroutine:

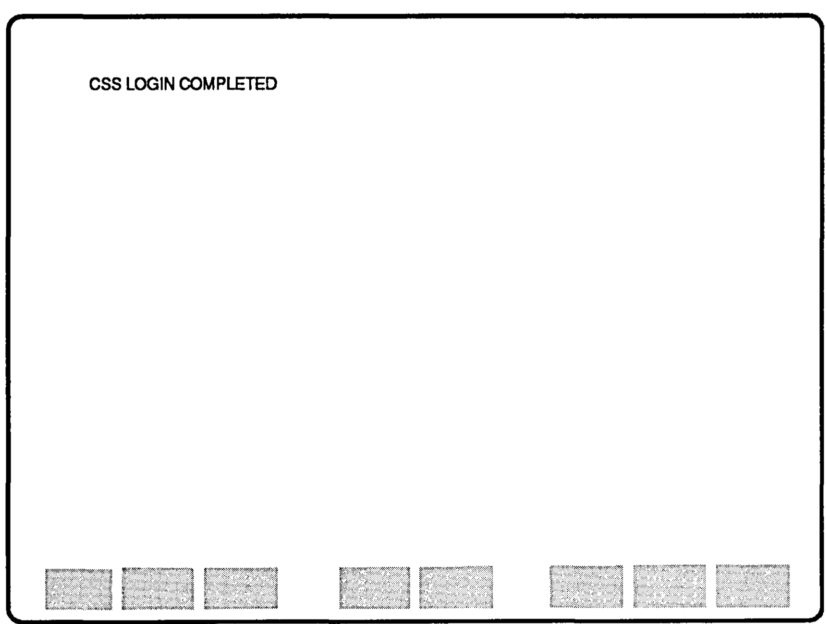



Note the use of the **BTAB** and **TAB** commands. These commands ensure that data is entered in the correct fields when the length of data entered would cause automatic skipping.

5 Waits for the synchronous host application screen and enters the user identification and password data. The following screen shows sample data automatically entered by the subroutine:



**6** Waits for the following *LOGIN COMPLETED* screen:




If the *loginims* subroutine logs in to the application successfully, the return code is set to 0; an unsuccessful login sets the return code to 1.

## Login Failed

The main program checks the value of the *rtncode* variable set in the *loginims* subroutine. If the value of *rtncode* is not 0, the program logs off from the synchronous host and waits for the sign-on screen. The local session, L1, is activated and the *login* local screen is displayed.

The *status* field is initially defined as Protected to prevent users from writing to this area. The attributes for this field are changed to Unprotected to allow the main program to write the *Login Failed* error message to the *status* field. The main program

immediately changes the *status* field attributes back to Protected after the error message is written.

```
if (rtncode != 0)      /* did log in fail?
  then
    clear              /* log off IMS
    text "/rcl"
    call aid_resp (0)
    while !($scan("WELCOME")) /* wait for sign-on screen
      do
        fresh
      endo

    connect (L1)
    fldpos = $fldaddr(login.status)
    chgattr (L1, fldpos, (U,*,H,*,R,*,*))
    if rtncode = 2
      then
        login.status = ("Host Login Failed.  " +
                        "System Not Available.")
      else
        login.status = ("Host Login Failed.  " +
                        "Please Verify Login Parameters.")
    endif
    chgattr (L1, fldpos, (P,*,H,*,R,*,*))
    home
  else
    clear
  endif
endo
```

The following shows the *login* local screen and the error message:

```
                        HOST LOGIN SCREEN
Application Id :   imssys 1
RACF User Id :     cssuser 1
RACF Password :
CSS User Id :      cssid 1
CSS Password :










Host Login Failed. Please Verify Login Parameters.
```

# Add New Customer

## Enter Data

Following successful login to the synchronous host session, the *addcust* screen is activated and displayed:

CUSTOMER ADD SCREEN

Service Branch :
Name :
Street :
City : State : Zip Code :
Phone No. :

Press PF12 to EXIT.

The program exits to Tutorial mode to allow the user to enter the appropriate customer data. The program terminates and logs off from the synchronous host session if PF12 is pressed.

```
connect (L2)                    /* activate local session 2

while (1)
  do
    show (L2)                   /* display local session 2
    exit                        /* exit to tutorial

    if (sysaid = 12)             /* exit addcust, log off IMS
      then
        break
    endif
```

The screen below shows the *CUSTOMER ADD SCREEN* after data has been entered by the user. Note that the zip code entered, *07601*, is incorrect for the customer's city, *Red Bank*.

```
                         CUSTOMER ADD SCREEN
Service Branch :  ussonlee
Name :   John Robinson
Street : 123 Main Street
City :   Red Bank                 State : NJ    Zip Code : 07601
Phone No. : 201 555 1234


Press PF12 to EXIT.
```

## Assign Data

When program execution is resumed, after the customer data is entered and (RETURN) is pressed, the program saves the data entered via the *addcust* local screen format by assigning the data to global variables for later use.

```
format addcust
branch   = .branch
name     = .name
street   = .street
geocd    = "          "
city     = .city
state    = .state
zip      = .zip
areacd   = .areacd
nnx      = .nnx
exch     = .exch
```

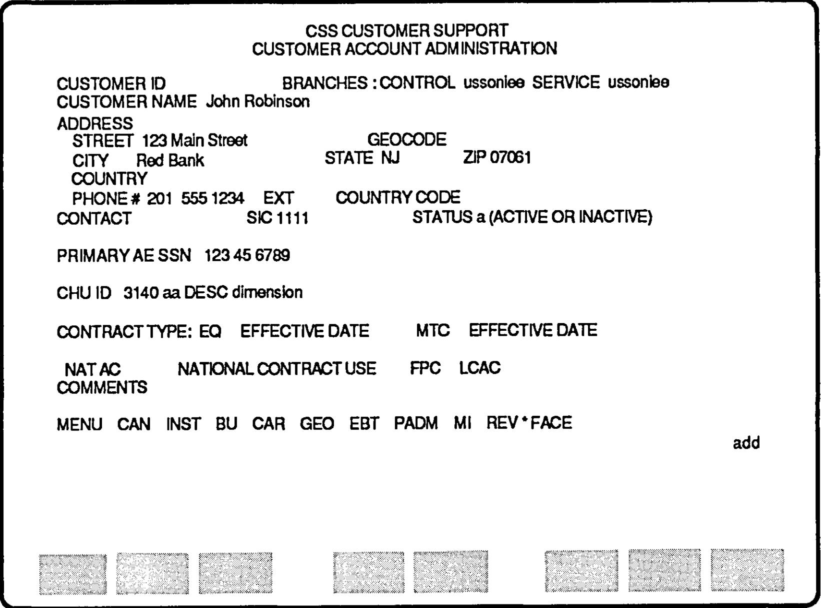
# Update Synchronous Host

## Populate Host Fields

The main program activates and displays the synchronous host session. When the host application screen is displayed, the *addcust* program calls the *popuflds* script. Review the *popuflds* script listed within the *addcust* program at the end of this section. The *popuflds* script assigns data entered from the local screen, together with hard-coded values, to the host field variables detailed in the *custadd* host screen format file.

```
connect (H1)                    /* activate host session
show (H1)                       /* display host session

call aid_resp(25)
text ("/for custadd")
call aid_resp(0)

tryagain = "y"
while (tryagain = "y")          /* ok to add customer
 do
  call popuflds                 /* populate host fields
  call aid_kc(4)

  tryagain = "n"                /* init to good ending first
```

The following screen shows the host application populated with data:



### Successful Update

The main program scans the host application screen for the successful update message and, if found, activates the *addcust* local screen format and displays a suitable message:

```
if $scan("ADD COMPLETE"(24,1,80))  /* success?
 then
  connect (L2)
  addcust.zip = zip
  fldpos = $fldaddr(addcust.status)
  chgattr (L2, fldpos, (U,*,H,*,R,*,*))
  addcust.status = "CUSTOMER ADD SUCCESSFUL."
  chgattr (L2, fldpos, (P,*,H,*,R,*,*))
  home
  break
endif
```

### Zip Code Error Routine

In this example, the zip code entered does not match the customer's city and state. The main program scans the host application screen for the relative fail message and calls the *fixzip* script. Refer to the *fixzip* script within the *addcust.p* program at the end of this section.

The *fixzip* script uses the host screen format file *chkzip.f*, also listed at the end of this section, to access a host zip code reference screen to retrieve the correct zip code.

If the host process is unable to correct the error, the program activates the *addcust* local screen format and displays a suitable message:

```
    if $scan("INVALID ZIP WITHIN STATE"(24,1,80))
      then
       call fixzip       /* try to fix zip code
       clear
       text ("/for custadd")
       call aid_resp(0)
      else
       connect (L2)
       addcust.zip = zip
       fldpos = $fldaddr(addcust.status)
       chgattr (L2, fldpos, (U,*,H,*,R,*,*))
       addcust.status = "CUSTOMER ADD FAILED."
       chgattr (L2, fldpos, (P,*,H,*,R,*,*))
       home
    endif
  endo
endo
```

In this example, the host process is able to correct the zip code. The corrected zip code is stored in a global variable, the program repopulates the host field variables and adds the customer information to the data base. The following shows the *addcust* local screen and message after the customer information has been added to the data base:

```
                    CUSTOMER ADD SCREEN
Service Branch :  ussoniee
Name :     John Robinson
Street :   123 Main Street
City :     Red Bank              State : NJ    Zip Code : 07701
Phone No. :  201  555  1234

Press PF12 to EXIT.

CUSTOMER ADD SUCCESSFUL
```

Note that the zip code field has been amended by the host process and the correct zip code, *07701*, is automatically entered in the relative local screen field.

## Log off from Synchronous Host

The next section of the *addcust* program reactivates and redisplays the synchronous host session and automatically enters the IMS log off procedure.

```
        connect (H1)                    /* activate host session
        show (H1)                       /* display host session
        call aid_resp(25)
        text ("/rcl")
        call aid_resp(0)
ends
```

## Copy Subroutines

The final section of the *addcust* program uses the **COPY** preprocessor command to copy the various subroutines and the *loginims.s* file:

```
copy "/usr/escort/slib/aid_resp"
copy "/usr/escort/slib/aid_kc"
copy "/loginims.s"
```

## Program and File Listings

This section contains program listings for the *addcust.p* program and the *loginims.s* subroutine; the local screen format files, *addcust.l* and *login.l*; and the host screen format files, *custadd.f* and *chkzip.f*.

# addcust.p Program

```
/*************************************************************************
/*                                                                       *
/*                          ADD_CUST                                     *
/*                                                                       *
/*************************************************************************
add_cust  prog  main (L1)
          /***********************************/
          /*  GLOBAL VARIABLE DECLARATIONS    */
          /***********************************/

          /* Host Log in Variables */

          char(8) applic              /* host application id
          char(8) racfid              /* RACF User id
          char(8) racfpwd             /* RACF User password
          char(8) cssid               /* application User id
          char(8) csspwd              /* application User password

          /* Customer Info Variables */

          char(8)     branch
          char(40)    name
          char(30)    street
          char(9)     geocd
          char(15)    city
          char(2)     state
          char(5)     zip
          char(5)     areacd
          char(3)     nnx
          char(4)     exch

          /* Miscellaneous Variables */

          int         fldpos
          char(1)     tryagain

          /***********************************/
          /*  LOCAL SCREEN FORMAT DEFINITIONS */
          /***********************************/

          copy    "addcust.l"         /* customer information screen
          copy    "login.l"           /* login parameters screen

          /*************************************/
          /*  HOST SCREEN FORMAT DEFINITIONS    */
          /*************************************/

          copy    "custadd.f"         /* customer administration screen
          copy    "chkzip.f"         /* zip code screen
/************************** MAIN SCRIPT ******************************/

main      script
          int  rtncode                /* subroutine return code

          /**********************************************************/
          /* Set Up Local Sessions 1:(Login Parameters) 2:(Customer Info) */
          /**********************************************************/

          getfmt (L1, login)          /* assoc local scrn fmt with L1
          getfmt (L2, addcust)        /* assoc local scrn fmt with L2

          rtncode = 1

          while (rtncode != 0)        /* while log in failed
            do
```

```
show (L1)              /* display local session 1
exit                   /* exit to tutorial mode

/****************************************/
/* Assign Log in Parameters to Variables */
/****************************************/

format login
applic = .applic        /* host application id
racfid = .racfid
racfpwd = .racfpwd
cssid = .cssid
csspwd = .csspwd

/**************************************/
/* Log in to IMS Host Application      */
/**************************************/

connect (H1)           /* activate host session 1
if (sysret = -1)
then
  connect (L1)
  fldpos = $fldaddr(login.status)
  chgattr (L1, fldpos, (U,*,H,*,R,*,*))
  login.status = ("Host System Not Available.")
  chgattr (L1, fldpos, (P,*,H,*,R,*,*))
  rtncode = 2
  cycle
endif
show (H1)              /* display host session 1

while !($scan("WELCOME")) /* wait for sign-on screen
  do
    fresh
  endo

call loginims(applic,
              racfid,
              racfpwd,
              cssid,
              csspwd,
              rtncode)

if (rtncode != 0)       /* did log in fail?
  then
    clear              /* log off IMS
    text "/rcl"
    call aid_resp (0)
    while !($scan("WELCOME")) /* wait for sign-on screen
      do
        fresh
      endo

    connect (L1)
    fldpos = $fldaddr(login.status)
    chgattr (L1, fldpos, (U,*,H,*,R,*,*))
    if rtncode = 2
      then
        login.status = ("Host Login Failed.  " +
                        "System Not Available.")
      else
        login.status = ("Host Login Failed.  " +
                        "Please Verify Login Parameters.")
    endif
    chgattr (L1, fldpos, (P,*,H,*,R,*,*))
    home
  else
    clear
```

```
    endif
  endo
/****************************/
/* Log in to IMS successful */
/****************************/

connect (L2)                      /* activate local session 2

while (1)
  do
    show (L2)                     /* display local session 2
    exit                          /* exit to tutorial

    if (sysaid = 12)               /* exit addcust, log off IMS
      then
        break
    endif

    /*********************************************/
    /* Assign Customer Information to Variables */
    /*********************************************/

    format addcust
    branch    = .branch
    name      = .name
    street    = .street
    geocd     = "          "
    city      = .city
    state     = .state
    zip       = .zip
    areacd    = .areacd
    nnx       = .nnx
    exch      = .exch

    /*************************************************/
    /* Update Host DB with Customer Information      */
    /*************************************************/

    connect (H1)                     /* activate host session
    show (H1)                        /* display host session

    call aid_resp(25)
    text ("/for custadd")
    call aid_resp(0)

    tryagain = "y"
    while (tryagain = "y")            /* ok to add customer
     do
      call popuflds                  /* populate host fields
      call aid_kc(4)

      tryagain = "n"                  /* init to good ending first

      if $scan("ADD COMPLETE"(24,1,80))  /* success?
       then
        connect (L2)
        addcust.zip = zip
        fldpos = $fldaddr(addcust.status)
        chgattr (L2, fldpos, (U,*,H,*,R,*,*))
        addcust.status = "CUSTOMER ADD SUCCESSFUL."
        chgattr (L2, fldpos, (P,*,H,*,R,*,*))
        home
        break
      endif

      /*******************/
      /* If bad zip code */
      /*******************/

      if $scan("INVALID ZIP WITHIN STATE"(24,1,80))
```

```
          then
           call fixzip       /* try to fix zip code
           clear
           text ("/for custadd")
           call aid_resp(0)
          else
           connect (L2)
           addcust.zip = zip
           fldpos = $fldaddr(addcust.status)
           chgattr (L2, fldpos, (U,*,H,*,R,*,*))
           addcust.status = "CUSTOMER ADD FAILED."
           chgattr (L2, fldpos, (P,*,H,*,R,*,*))
           home
        endif
      endo
    endo
  /********************/
  /* Log off from IMS */
  /********************/

  connect (H1)                   /* activate host session
  show (H1)                      /* display host session

  call aid_resp(25)
  text ("/rcl")
  call aid_resp(0)

ends
```

```
/******************** POPUFLDS SCRIPT ************************************/
/* The purpose of this script is to populate all the necessary fields on */
/* host screen in order to add a customer into the host.                 */
/* These fields will be derived from what was entered in the local       */
/* session in addition to hard-coded values.                             */
/*************************************************************************/

popuflds  script
          format custadd
          .clctrbr = branch
          .clsvcbr = branch
          .clname  = name
          .clstr   = street
          .clgeo   = "          "
          .clcity  = city
          .clstate = state
          .clzip   = zip
          .clarcd  = areacd
          .clnnx   = nnx
          .clexch  = exch
          .clsic   = "1111"
          .clstat  = "a"
          .claess1 = "123"
          .claess2 = "45"
          .claess3 = "6789"
          .clpricel= "nat"
          .clcmu   = "3140aa"
          .cldesc  = "dimension"
ends
/*************************************************************************/
```

```
/******************** FIXZIP SCRIPT ***************************************/
/* The purpose of this script is to determine the valid zip code for the */
/* city and state specified on the local screen.                         */
/*************************************************************************/

fixzip   script

         clear
         text ("/for chkzip")
         call aid_resp(0)

         chkzip.city = city
         chkzip.state = state
         call aid_kc(1)
         if $scan("COMPLETE"(24,1,80))
         then
           geocd = chkzip.rgeoco01      /* save the found geo code
           zip   = chkzip.rzipmi01      /* save the found matching zip
           tryagain = "y"
         else
         endif

ends
/*************************************************************************/
```

```
        copy "/usr/escort/slib/aid_resp"
        copy "/usr/escort/slib/aid_kc"
        copy "/loginims.s"
/***************************************************************************/
endp
```

# loginims.s Subroutine

```
/***********************************************************************
/*                                                                     *
/*                        LOGINIMS.S                                   *
/*                                                                     *
/***********************************************************************
/*                                                                     *
/*   FUNCTIONAL DESCRIPTION:                                           *
/*      log in to IMS                                                  *
/*   INPUT PARAMETERS:                                                 *
/*      applic   -> host application id                                *
/*      racfid   -> racf user id                                       *
/*      racfpwd  -> racf user password                                 *
/*      cssid    -> css user id                                        *
/*      csspwd   -> css user password                                  *
/*   OUTPUT PARAMETERS:                                                *
/*      rtncode  -> 0 = successful log in                              *
/*                  1 = log in rejected                                *
/*                  2 = system not available                           *
/*                                                                     *
/***********************************************************************

loginims script (char(8) appl,
                 char(8) usrid,
                 char(8) usrpwd,
                 char(8) csid,
                 char(8) cspwd,
                 int rtncode)

         text appl                          /* load RACF application
         enter

         while !($scan("USER ID"))          /* wait for sign-on screen
         do
           if ($scan("BOUND"))              /* check for system down
           then
             rtncode = 2                     /* set return code
             log screen                     /* save screen image
             return                         /* quit login script
           endif
           fresh
         endo
/**************** HOST LOG IN PROCEDURE ************************

         text usrid              /* load userid on screen
         btab                    /* position to start of field
         tab                     /* go to start of next field

         text usrpwd             /* load user password
         btab                    /* position to start of field
         tab                     /* go to start of next field

         text ("cssgrpl")       /* load system password on screen

         call aid_resp(0)        /* hit enter to log in to IMS

         clear

         text ("/for custlog")
         call aid_resp(0)

         while !($scan("PASSWORD")) /*wait for sign-on screen
         do
           fresh
         endo
```

```
        text csid              /* load userid on screen
        btab                   /* position to start of field
        tab                    /* go to start of next field
        text ("cssgrpl")       /* load system password on screen
        btab                   /* position to start of field
        tab                    /* go to start of next field
        text cspwd             /* load user password
        call aid_resp(0)       /* hit enter to log in to CSS
        if $scan("LOGIN COMPLETED") /* check for log in completed
        then
          rtncode = 0           /* set good return code
        else
          rtncode = 1           /* set log in rejected return code
        endif
endlog: ends                   /* end of log in script
```

# addcust.l Local Screen Format File

```
begfmt addcust
      field (1,30,20,(P,A,H,R,R,7,0)) DUMMY "CUSTOMER ADD SCREEN"
      field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "Service Branch: "
      field (3,22,8, (U,A,H,R,R,7,0)) addcust.branch
      field (5,5,9,  (P,A,H,R,N,7,0)) DUMMY "Name:    "
      field (5,15,35,(U,A,H,R,R,7,0)) addcust.name
      field (7,5,9,  (P,A,H,R,N,7,0)) DUMMY "Street:  "
      field (7,15,35,(U,A,H,R,R,7,0)) addcust.street
      field (9,5,7,  (P,A,H,R,N,7,0)) DUMMY "City: "
      field (9,15,15,(U,A,H,R,R,7,0)) addcust.city
      field (9,32,7, (P,A,H,R,N,7,0)) DUMMY "State: "
      field (9,40,2, (U,A,H,R,R,7,0)) addcust.state
      field (9,44,10,(P,A,H,R,N,7,0)) DUMMY "Zip Code: "
      field (9,55,5, (U,A,H,R,R,7,0)) addcust.zip
      field (11,5,12,(P,A,H,R,N,7,0)) DUMMY "Phone No.: "
      field (11,18,3,(U,A,H,R,R,7,0)) addcust.areacd
      field (11,23,3,(U,A,H,R,R,7,0)) addcust.nnx
      field (11,28,4,(U,A,H,R,R,7,0)) addcust.exch
      field (15,5,21,(P,A,H,R,N,7,0)) DUMMY " Press PF12 to EXIT. "
      field (24,2,70,(P,A,H,R,N,7,0)) addcust.status
endfmt
```

## login.l Local Screen Format File

```
begfmt login
       field (1,28,23,(P,A,H,R,R,7,0)) DUMMY "HOST LOGIN SCREEN"
       field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "Application Id: "
       field (3,22,8, (U,A,H,R,R,7,0)) login.applic
       field (5,5,16, (P,A,H,R,N,7,0)) DUMMY "RACF User Id:   "
       field (5,22,8, (U,A,H,R,R,7,0)) login.racfid
       field (7,5,16, (P,A,H,R,N,7,0)) DUMMY "RACF Password:  "
       field (7,22,8, (U,A,D,R,N,7,0)) login.racfpwd
       field (9,5,16, (P,A,H,R,N,7,0)) DUMMY "CSS User Id:    "
       field (9,22,8, (U,A,H,R,R,7,0)) login.cssid
       field (11,5,16,(P,A,H,R,N,7,0)) DUMMY "CSS Password:   "
       field (11,22,8,(U,A,D,R,N,7,0)) login.csspwd
       field (24,2,70,(P,A,H,R,N,7,0)) login.status
endfmt
```

# custadd.f Host Screen Format File

```
field (04,48,0008)  custadd.clctrbr
field (04,66,0008)  custadd.clsvcbr
field (05,15,0040)  custadd.clname
field (08,10,0030)  custadd.clstr
field (08,51,0009)  custadd.clgeo
field (09,10,0020)  custadd.clcity
field (09,41,0002)  custadd.clstate
field (09,56,0010)  custadd.clzip
field (11,11,0005)  custadd.clarcd
field (11,17,0003)  custadd.clnnx
field (11,21,0004)  custadd.clexch
field (12,34,0004)  custadd.clsic
field (12,59,0001)  custadd.clstat
field (14,17,0003)  custadd.claess1
field (14,21,0002)  custadd.claess2
field (14,24,0004)  custadd.claess3
field (14,72,0003)  custadd.clprice1
field (16,09,0006)  custadd.clcmu
field (16,21,0030)  custadd.cldesc
```

# chkzip.f Host Screen Format File

```
field (04,27,0015)  chkzip.city
field (04,19,0002)  chkzip.state
field (05,07,0005)  chkzip.rzipmi01
field (05,65,0009)  chkzip.rgeoco01
```

# Asynchronous Host Sample Program

This sample program provides a new front-end for users who are responsible for adding customer information to an asynchronous host data base. The program performs the login procedure, prompts for customer information, takes corrective action if the user enters an invalid zip code, and logs the user off when necessary.

This sample program is similar to the synchronous host sample program provided in this chapter. Compare the two samples to review the differences in the code necessary in the two environments.

The sample comprises two scripts, a main program, *addcust.ap* and a subroutine, *login.s*; together with two local screen format files, *addcust.l* and *login.l*; and two host screen format files, *custadd.f* and *chkzip.f*.

The login subroutine, *login.s*, accepts login information from the user via a local session screen and automates the asynchronous host login procedure. The main program, *addcust.ap*, takes customer information, entered by the user in a local session screen, and updates an asynchronous host data base.

Execution of the program is subdivided into six main sections:

- Declaration of variables and definition of screen formats.
- Log in to asynchronous host application.
- Collect new customer information.
- Update asynchronous host data base with new customer information.
- Log off from host.
- Copy subroutines.

Four error conditions are demonstrated:

- Failure to log in to host.
- Line drop.
- Time out.
- Zip code entered does not match customer's city and state.

# Program Execution

## Declarations and Definitions

The first section of the *addcust.ap* program comprises the Global Variable Declarations section and the Local and Host Screen Format Definition sections.

### Global Variable Declarations

The three subsections in the Global Variables declaration section declare variables for use with the asynchronous host login and customer information procedures, and for general use.

Two host login variables are declared, both 8 characters in length:

```
char(8) userid                /* application User id
char(8) userpwd               /* application User password
```

Ten customer information variables are declared with the character lengths indicated:

```
char(8)    branch
char(40)   name
char(30)   street
char(9)    geocd
char(15)   city
char(2)    state
char(5)    zip
char(5)    areacd
char(3)    nnx
char(4)    exch
```

Two miscellaneous variables are declared:

```
int        fldpos
char(1)    tryagain
```

### Local Screen Format Definitions

The two local screen formats, *addcust.l* and *login.l*, are defined in this section:

```
copy   "addcust.l"        /* customer information screen
copy   "login.l"          /* login parameters screen
```

The **COPY** preprocessor command inserts the content of the *addcust.l* and *login.l* local screen format files into the main program.

Both local screen format files use the *DUMMY* keyword and a literal to produce field narratives on the screen. The attribute lists define certain fields as Protected or Unprotected, and reverse video or normal display. Refer to the **FIELD** command in

Chapter 4 for a complete list of definable attributes.

The *addcust.l* local screen format has the defined screen name, addcust:

```
begfmt addcust
       field (1,30,20,(P,A,H,R,R,7,0)) DUMMY "CUSTOMER ADD SCREEN"
       field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "Service Branch: "
       field (3,22,8, (U,A,H,R,R,7,0)) addcust.branch
       field (5,5,9,  (P,A,H,R,N,7,0)) DUMMY "Name:    "
       field (5,15,35,(U,A,H,R,R,7,0)) addcust.name
       field (7,5,9,  (P,A,H,R,N,7,0)) DUMMY "Street:  "
       field (7,15,35,(U,A,H,R,R,7,0)) addcust.street
       field (9,5,7,  (P,A,H,R,N,7,0)) DUMMY "City: "
       field (9,15,15,(U,A,H,R,R,7,0)) addcust.city
       field (9,32,7, (P,A,H,R,N,7,0)) DUMMY "State: "
       field (9,40,2, (U,A,H,R,R,7,0)) addcust.state
       field (9,44,10,(P,A,H,R,N,7,0)) DUMMY "Zip Code: "
       field (9,55,5, (U,A,H,R,R,7,0)) addcust.zip
       field (11,5,12,(P,A,H,R,N,7,0)) DUMMY "Phone No.: "
       field (11,18,3,(U,A,H,R,R,7,0)) addcust.areacd
       field (11,23,3,(U,A,H,R,R,7,0)) addcust.nnx
       field (11,28,4,(U,A,H,R,R,7,0)) addcust.exch
       field (15,5,21,(P,A,H,R,N,7,0)) DUMMY " Press F8 to EXIT. "
       field (24,2,70,(P,A,H,R,N,7,0)) addcust.status
endfmt
```

The *login.l* local screen format has the defined screen name, login. Note that this local screen format makes use of the non-displayable (dark) attribute for the password field:

```
begfmt login
       field (1,28,23,(P,A,H,R,R,7,0)) DUMMY "HOST LOGIN SCREEN"
       field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "User Id:    "
       field (3,22,8, (U,A,H,R,R,7,0)) login.userid
       field (5,5,16,(P,A,H,R,N,7,0)) DUMMY "Password:    "
       field (5,22,8,(U,A,D,R,N,7,0))  login.userpwd
       field (24,2,70,(P,A,H,R,N,7,0)) login.status
endfmt
```

## Host Screen Format Definitions

Two host screen formats are also inserted into the main program by the following COPY commands:

```
copy   "custadd.f"        /* customer administration screen
copy   "chkzip.f"         /* zip code screen
```

The listings for these two files are shown at the end of this section.

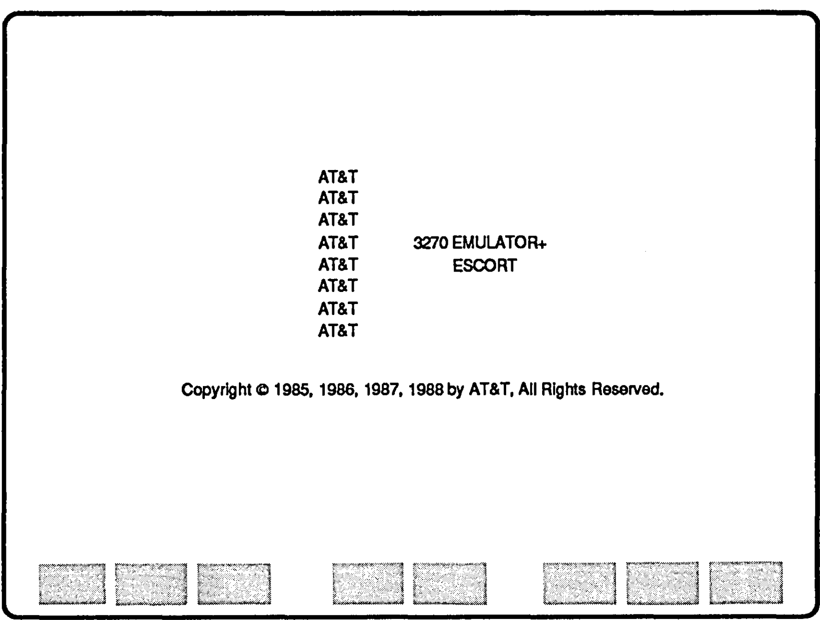
# Log in to Asynchronous Host

## Execute Main Program

To run this sample program, at the UNIX shell prompt, the user types

***escort addcust.ap***

and presses RETURN. The following ESCORT banner screen is displayed briefly:




The first program line

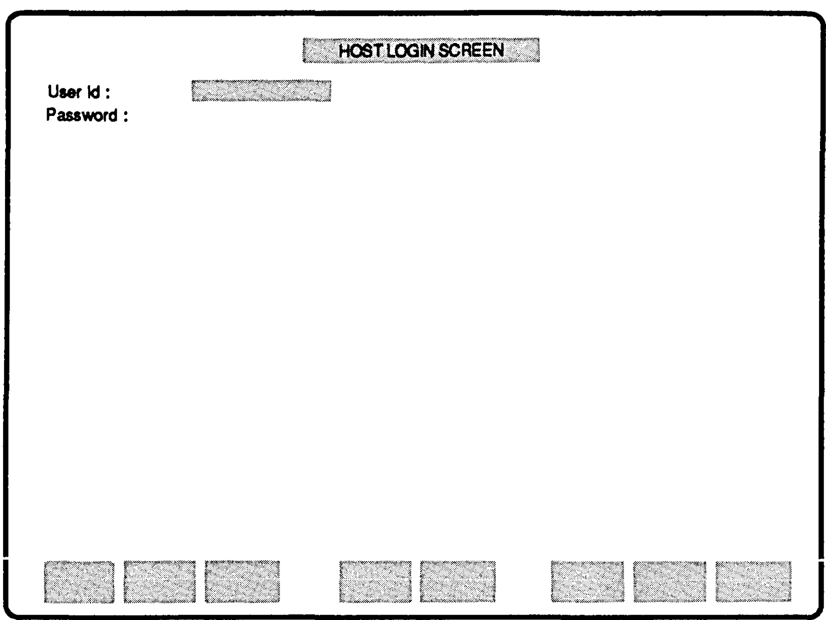
```
add_cust  prog  main (L1)
```

indicates the beginning of the program, identifies the first script named *main* and connects to local session, L1. A local variable is declared in the first script:

```
main       script
           int  rtncode               /* subroutine return code
```

## Load Formats and Enter Data

The next section of program loads the two local screen formats, *login.l* and *addcust.l* into local sessions, L1 and L2 respectively. The *login.l* screen is activated and displayed:




The program exits to Tutorial mode to allow the user to enter the appropriate login data:

```
getfmt (L1, login)          /* assoc local scrn fmt with L1
getfmt (L2, addcust)        /* assoc local scrn fmt with L2

rtncode = 1

while (rtncode != 0)         /* while log in failed
  do

    show (L1)               /* display local session 1
    exit                    /* exit to tutorial mode
```

### Assign Data

When program execution is resumed, after the login data is entered and RETURN is pressed, the program saves the data entered via the *login* local screen format by assigning the data to global variables for later use. The **FORMAT** command defines the default format as *login*; it is not necessary, therefore, to assign the format name to individual fields.

```
format login
userid = .userid
userpwd = .userpwd
```

### Activate Asynchronous Host Session

The main program activates asynchronous host session A1. The value of the system global variable, *SYSRET*, is checked to determine whether the connection to the host is successful.

If the connection failed, the attributes for the *status* field, initially defined as Protected to prevent users from writing to this area, are changed to Unprotected to allow the program to write the *System Not Available Connect Failed* error message to the *status* field. The *status* field attributes are changed back to Protected after the error message is written.

```
serinit (1,1200,e,1,7,full,"5551234","")
connect (A1)          /* activate host session 1
if (sysret = -1)
  then
    connect (L1)
    fldpos = $fldaddr(login.status)
    chgattr (L1, fldpos, (U,*,H,*,R,*,*))
    login.status = ("System Not Available.  " +
                    "Connect Failed.")
    chgattr (L1, fldpos, (P,*,H,*,R,*,*))
    rtncode = 2
    cycle
endif
show (A1)             /* display host session 1
```

The *addcust.ap* program calls the *login* subroutine and passes three parameters:

```
call login(userid,
           userpwd,
           rtncode)
```

## Log in Procedure

Review the program listing for the *login.s* subroutine at the end of this section. The **CALL** command invokes the *login* subroutine to log in to the application. The *login* **SCRIPT** statement has a declaration list corresponding to the parameter list in the *addcust* **CALL** command.

The *login* subroutine returns one of three codes to the *addcust* program via the *rtncode* variable:

| | |
|---|---|
| 0 | Successful login. |
| 1 | Login rejected. |
| 2 | System not available.. |

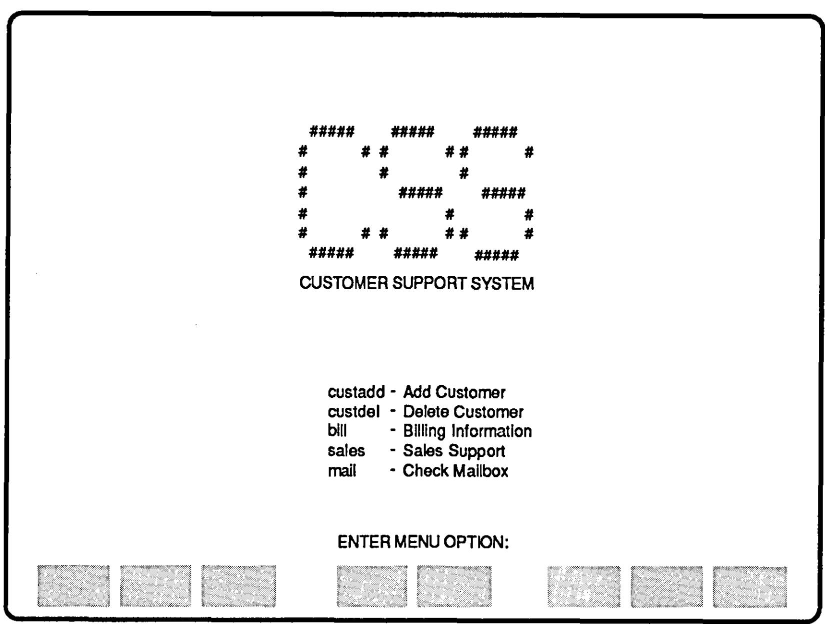
The *login* subroutine

1. Waits for the login prompt.
2. Checks for system failure. If the asynchronous host system is not available, the return code is set to 2 and the *login* local screen is redisplayed.

3 Enters login information. The following screen shows the login data automatically entered by the ESCORT subroutine:

```
LOGIN: abc
USER ID: user1
PASSWORD:
```

4 Waits for the following asynchronous host menu screen:




If the *login* subroutine logs in to the application successfully, the return code is set to 0; an unsuccessful login sets the return code to 1.

### Login Failed

The main program checks the value of the *rtncode* variable set in the *login* subroutine. If the value of *rtncode* is not 0, the program disconnects from the asynchronous host. The local session, L1, is activated and the *login* local screen is displayed.

The *status* field is initially defined as Protected to prevent users from writing to this area. The attributes for this field are changed to Unprotected to allow the main program to write the *Login Failed* error message to the *status* field. The main program immediately changes the *status* field attributes back to Protected

after the error message is written.

```
if (rtncode != 0)      /* did log in fail?
  then
    discon (A1)

    connect (L1)
    fldpos = $fldaddr(login.status)
    chgattr (L1, fldpos, (U,*,H,*,R,*,*))
    if rtncode = 2
      then
        login.status = ("Host Login Failed.  " +
                        "System Not Available.")
      else
        login.status = ("Host Login Failed.  " +
                        "Please Verify Login Parameters.")
    endif
    chgattr (L1, fldpos, (P,*,H,*,R,*,*))
    home
  endif
endo
```

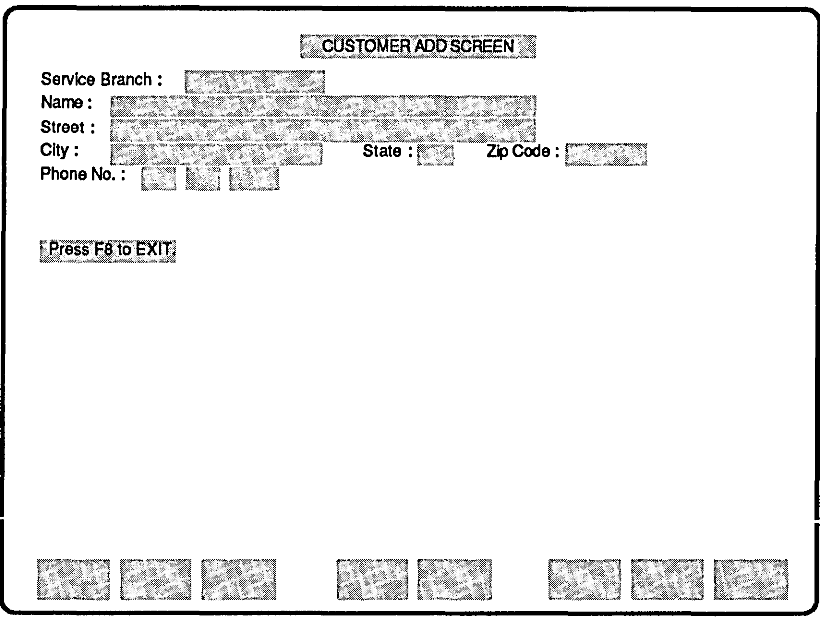
The following shows the *login* local screen and the error message:

```
                    HOST LOGIN SCREEN
User Id :        user1
Password :




 Host Login Failed.  Please Verify Login Parameters.
```

# Add New Customer

## Enter Data

Following successful login to the asynchronous host session, the *addcust* local screen is activated and displayed:




The program exits to Tutorial mode to allow the user to enter the appropriate customer data. The program terminates and logs off from the asynchronous host session if F8 is pressed.

```
connect (L2)                    /* activate local session 2

while (1)
  do
    show (L2)                   /* display local session 2
    exit                        /* exit to tutorial

    if (sysaid = 8)              /* exit addcust, log off
      then
        break
    endif
```

The screen below shows the *CUSTOMER ADD SCREEN* after data has been entered by the user. Note that the zip code entered, *07601*, is incorrect for the customer's city, *Red Bank*.

```
                      CUSTOMER ADD SCREEN
Service Branch :   ussonlee
Name :   John Robinson
Street : 123 Main Street
City :   Red Bank              State : NJ   Zip Code : 07601
Phone No. : 201 555 1234


Press F8 to EXIT.
```

## Assign Data

When program execution is resumed, after the customer data is entered and (RETURN) is pressed, the program saves the data entered via the *addcust* local screen format by assigning the data to global variables for later use.

```
format addcust
branch   = .branch
name     = .name
street   = .street
geocd    = "          "
city     = .city
state    = .state
zip      = .zip
areacd   = .areacd
nnx      = .nnx
exch     = .exch
```

## Update Asynchronous Host

### Populate Host Fields

The main program activates and displays the asynchronous host session. The main program calls the *send_aid* script. The *send_aid* script is listed within the *addcust.ap* program at the end of this section.

The *send_aid* subroutine sends a specified soft function key to the host. The subroutine scans the asynchronous data received, using a **WAIT** command and the strings passed to it as parameters in the **CALL** statement, and returns one of four values to the main script via the *SYSRET* variable.

In this case, only one string is passed to the *send_aid* subroutine and, therefore, one of the following three values is returned to the main script via the *SYSRET* variable:

| | |
|---|---|
| 1 | The last line of the host screen, containing the string, *MI REV FACE*, detected. |
| −99 | The *LOGIN* prompt detected, line dropped. |
| −1 | **WAIT** command timed out. |

If the value of *SYSRET* is less than zero, that is, the line dropped or the **WAIT** command timed out, the main program calls the *err_msg* script. The *err_msg* script is listed within the *addcust.ap* program at the end of this section.

The *err_msg* subroutine activates the *addcust* local screen and tests the value of *SYSRET*. If the line is dropped or the script times out, the attributes for the *status* field, initially defined as Protected to prevent users from writing to this area, are changed to Unprotected to allow the program to write either the *Host Connection Failed Line Dropped* or the *Host Connection Failed Timed Out* error messages to the *status* field. The *status* field attributes are changed back to Protected after the error message is written and the *login* local screen is redisplayed.

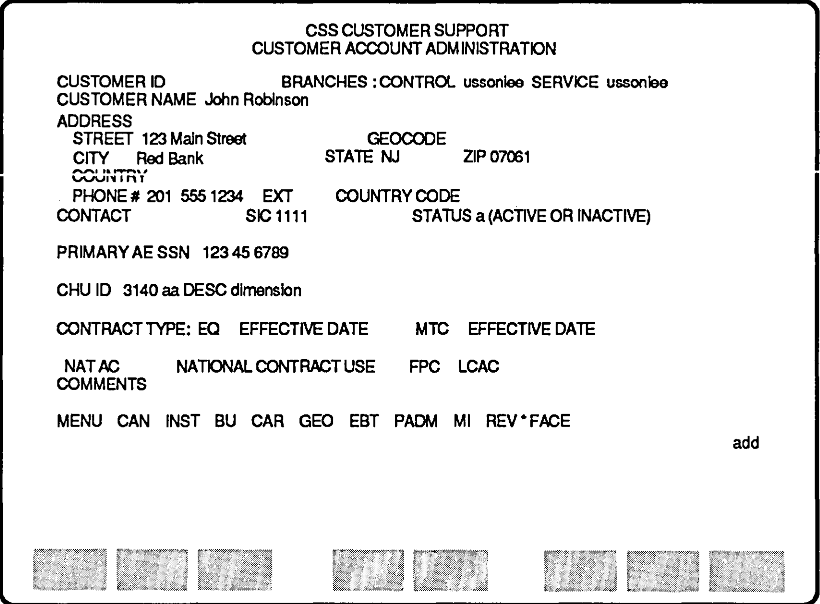
Line drops do not occur frequently, they are included in this sample program to demonstrate possible solutions to detect such problems.

When the host application screen is displayed, the *addcust*

program calls the *popuflds* script. Review the *popuflds* script listed within the *addcust* program at the end of this section. The *popuflds* script assigns data entered from the local screen, together with hard-coded values, to the host field variables detailed in the *custadd* host screen format file.

```
connect (A1)                      /* activate host session
show (A1)                         /* display host session

text "custadd"
call send_aid (0,"MI  REV  FACE","")
if (sysret < 0)
  then
    call err_msg
    break
endif

tryagain = "y"
while (tryagain = "y")             /* ok to add customer
  do
    call popuflds                   /* populate host fields
```

The following screen shows the host application populated with data.



### Successful Update

The main program calls the *send_aid* script. The *send_aid* subroutine returns one of the following four values to the main script via the *SYSRET* variable:

| | |
|---|---|
| 1 | Successful update message, *ADD COMPLETE*, detected. |
| 2 | *INVALID ZIP WITHIN STATE* error message detected. |
| −99 | The *LOGIN* prompt detected, line dropped. |
| −1 | **WAIT** command timed out. |

If the value of *SYSRET* is less than zero, the main program calls the *err_msg* script.

If the successful update message, *ADD COMPLETE*, is detected the main program activates the *addcust* local screen format and displays a suitable message:

```
call send_aid (4,
               "ADD COMPLETE",
               "INVALID ZIP WITHIN STATE")
if (sysret < 0)
  then
    call err_msg
    break
endif
tryagain = "n"                /* init to good ending first
if (sysret = 1)               /* success?
  then
    connect (L2)
    addcust.zip = zip
    fldpos = $fldaddr(addcust.status)
    chgattr (L2, fldpos, (U,*,H,*,R,*,*))
    addcust.status = "CUSTOMER ADD SUCCESSFUL."
    chgattr (L2, fldpos, (P,*,H,*,R,*,*))
    home
    break
endif
```

### Zip Code Error Routine

In this example, the zip code entered does not match the customer's city and state. The *send_aid* subroutine returns a *SYSRET* value of 2 and the main program calls the *fixzip* script. Refer to the *fixzip* script within the *addcust.ap* program at the end of this section.

The *fixzip* script uses the host screen format file *chkzip.f*, also listed at the end of this section, to access a host zip code reference screen to retrieve the correct zip code.

The main program again calls the *send_aid* script to go back to the *addcust* screen. The *send_aid* subroutine returns one of the following three values to the main script via the *SYSRET* variable:

| | |
|---|---|
| 1 | The last line of the host screen, containing the string, *MI REV FACE*, detected. |
| −99 | The *LOGIN* prompt detected, line dropped. |
| −1 | **WAIT** command timed out. |

If the value of *SYSRET* is less than zero, the main program calls the *err_msg* script.

If the host process is unable to correct the error, the program activates the *addcust* local screen format and displays a suitable message:

```
          if (sysret = 2)
            then
              call fixzip       /* try to fix zip code
              call send_aid (5,                /* go back to
                             "MI  REV  FACE",  /* addcust screen
                             "")
              if (sysret < 0)
                then
                  call err_msg
                  break
              endif
            else
              connect (L2)
              addcust.zip = zip
              fldpos = $fldaddr(addcust.status)
              chgattr (L2, fldpos, (U,*,H,*,R,*,*))
              addcust.status = "CUSTOMER ADD FAILED."
              chgattr (L2, fldpos, (P,*,H,*,R,*,*))
              home
          endif
      endo
    endo
```

In this example, the host process is able to correct the zip code. The corrected zip code is stored in a global variable, the program repopulates the host field variables and adds the customer information to the data base. The following shows the *addcust* local screen and message after the customer information has been added to the data base:

CUSTOMER ADD SCREEN

Service Branch : ussonlee

Name : John Robinson

Street : 123 Main Street

City : Red Bank State : NJ Zip Code : 07701

Phone No. : 201 555 1234

Press F8 to EXIT.

CUSTOMER ADD SUCCESSFUL

Note that the zip code field has been amended by the host process and the correct zip code, *07701*, is automatically entered in the relative local screen field.

## Log off from Asynchronous Host

The next section of the *addcust* program reactivates and redisplays the asynchronous host session and calls the *send_aid* script. The *send_aid* subroutine waits for the *ENTER MENU OPTION* prompt. The main program logs off from the asynchronous host session.

```
        connect (A1)                     /* activate host session
        show (A1)                        /* display host session
        call send_aid (8,"ENTER MENU OPTION","")
        text "exit"
        enter
        discon (A1)
ends
```

## Copy Subroutines

The final section of the *addcust* program uses the **COPY** preprocessor command to copy the *login.s* file:

```
copy "/login.s"
```

## Program and File Listings

This section contains program listings for the *addcust.ap* program and the *login.s* subroutine; the local screen format files, *addcust.l* and *login.l*; and the host screen format files, *custadd.f* and *chkzip.f*.

# addcust.ap Program

```
/*****************************************************************************
/*                                                                           *
/*                          ADD_CUST                                         *
/*                                                                           *
/*****************************************************************************

add_cust  prog  main (L1)
          /************************************/
          /*  GLOBAL VARIABLE DECLARATIONS    */
          /************************************/

          /* Host Log in Variables */

          char(8) userid              /* application User id
          char(8) userpwd             /* application User password

          /* Customer Info Variables */

          char(8)     branch
          char(40)    name
          char(30)    street
          char(9)     geocd
          char(15)    city
          char(2)     state
          char(5)     zip
          char(5)     areacd
          char(3)     nnx
          char(4)     exch

          /* Miscellaneous Variables */

          int         fldpos
          char(1)     tryagain

          /************************************/
          /*  LOCAL SCREEN FORMAT DEFINITIONS */
          /************************************/

          copy   "addcust.l"          /* customer information screen
          copy   "login.l"            /* login parameters screen

          /***************************************/
          /*  HOST SCREEN FORMAT DEFINITIONS     */
          /***************************************/

          copy   "custadd.f"          /* customer administration screen
          copy   "chkzip.f"          /* zip code screen
/************************** MAIN SCRIPT ********************************/

main      script
          int  rtncode                /* subroutine return code

          /***************************************************************/
          /* Set Up Local Sessions 1:(Login Parameters) 2:(Customer Info) */
          /***************************************************************/

          getfmt (L1, login)          /* assoc local scrn fmt with L1
          getfmt (L2, addcust)        /* assoc local scrn fmt with L2

          rtncode = 1

          while (rtncode != 0)         /* while log in failed
            do

              show (L1)               /* display local session 1
              exit                    /* exit to tutorial mode
```

```
    /*****************************************/
    /* Assign Log in Parameters to Variables */
    /*****************************************/

    format login
    userid = .userid
    userpwd = .userpwd

    /**********************************/
    /* Log in to Host Application     */
    /**********************************/

    serinit (1,1200,e,1,7,full,"5551234","")
    connect (A1)           /* activate host session 1
    if (sysret = -1)
      then
        connect (L1)
        fldpos = $fldaddr(login.status)
        chgattr (L1, fldpos, (U,*,H,*,R,*,*))
        login.status = ("System Not Available.  " +
                        "Connect Failed.")
        chgattr (L1, fldpos, (P,*,H,*,R,*,*))
        rtncode = 2
        cycle
    endif
    show (A1)              /* display host session 1

    call login(userid,
               userpwd,
               rtncode)

    if (rtncode != 0)       /* did log in fail?
      then
        discon (A1)

        connect (L1)
        fldpos = $fldaddr(login.status)
        chgattr (L1, fldpos, (U,*,H,*,R,*,*))
        if rtncode = 2
          then
            login.status = ("Host Login Failed.  " +
                            "System Not Available.")
          else
            login.status = ("Host Login Failed.  " +
                            "Please Verify Login Parameters.")
        endif
        chgattr (L1, fldpos, (P,*,H,*,R,*,*))
        home
    endif
  endo
/********************/
/* Log in successful */
/********************/

connect (L2)                    /* activate local session 2

while (1)
  do
    show (L2)                   /* display local session 2
    exit                        /* exit to tutorial

    if (sysaid = 8)              /* exit addcust, log off
      then
        break
    endif
```

```
/*********************************************/
/* Assign Customer Information to Variables */
/*********************************************/

format addcust
branch    = .branch
name      = .name
street    = .street
geocd     = "          "
city      = .city
state     = .state
zip       = .zip
areacd    = .areacd
nnx       = .nnx
exch      = .exch
/*************************************************/
/* Update Host DB with Customer Information      */
/*************************************************/

connect (A1)                    /* activate host session
show (A1)                       /* display host session

text "custadd"
call send_aid (0,"MI  REV  FACE","")
if (sysret < 0)
  then
    call err_msg
    break
endif

tryagain = "y"
while (tryagain = "y")          /* ok to add customer
  do
    call popuflds                 /* populate host fields

    call send_aid (4,
                   "ADD COMPLETE",
                   "INVALID ZIP WITHIN STATE")
    if (sysret < 0)
      then
        call err_msg
        break
    endif

    tryagain = "n"                /* init to good ending first

    if (sysret = 1)               /* success?
      then
        connect (L2)
        addcust.zip = zip
        fldpos = $fldaddr(addcust.status)
        chgattr (L2, fldpos, (U,*,H,*,R,*,*))
        addcust.status = "CUSTOMER ADD SUCCESSFUL."
        chgattr (L2, fldpos, (P,*,H,*,R,*,*))
        home
        break
    endif

    /******************/
    /* If bad zip code */
    /******************/

    if (sysret = 2)
      then
        call fixzip       /* try to fix zip code

        call send_aid (5,               /* go back to
                  "MI  REV  FACE",    /* addcust screen
```

```
                                   "")
                     if (sysret < 0)
                       then
                         call err_msg
                         break
                     endif
                   else
                     connect (L2)
                     addcust.zip = zip
                     fldpos = $fldaddr(addcust.status)
                     chgattr (L2, fldpos, (U,*,H,*,R,*,*))
                     addcust.status = "CUSTOMER ADD FAILED."
                     chgattr (L2, fldpos, (P,*,H,*,R,*,*))
                     home
                 endif
             endo
           endo
         /*********************/
         /* Log off from host */
         /*********************/
         connect (A1)                      /* activate host session
         show (A1)                         /* display host session
         call send_aid (8,"ENTER MENU OPTION","")
         text "exit"
         enter
         discon (A1)
ends
```

```
/******************** POPUFLDS SCRIPT ************************************/
/* The purpose of this script is to populate all the necessary fields on */
/* host screen in order to add a customer into the host.                 */
/* These fields will be derived from what was entered in the local       */
/* session in addition to hard-coded values.                             */
/*************************************************************************/
popuflds  script
          format custadd
          .clctrbr = branch
          .clsvcbr = branch
          .clname  = name
          .clstr   = street
          .clgeo   = "          "
          .clcity  = city
          .clstate = state
          .clzip   = zip
          .clarcd  = areacd
          .clnnx   = nnx
          .clexch  = exch
          .clsic   = "1111"
          .clstat  = "a"
          .claess1 = "123"
          .claess2 = "45"
          .claess3 = "6789"
          .clprice1= "nat"
          .clcmu   = "3140aa"
          .cldesc  = "dimension"
ends
/*************************************************************************/
```

```
/******************** FIXZIP SCRIPT *************************************/
/* The purpose of this script is to determine the valid zip code for the */
/* city and state specified on the local screen.                         */
/************************************************************************/

fixzip    script
          call send_aid (5,"CUST ADMIN","")
          if (sysret < 0)
            then
              call err_msg
              return
          endif
          chkzip.city = city
          chkzip.state = state
          call send_aid (1,"COMPLETE","")
          if (sysret < 0)
            then
              call err_msg
              return
            else
              geocd = chkzip.rgeoco01      /* save the found geo code
              zip   = chkzip.rzipmi01      /* save the found matching zip
              tryagain = "y"
          endif
ends
/************************************************************************/
```

```
/******************** SEND_AID SCRIPT ************************************/
/* The purpose of this script is to determine whether specified data     */
/* strings are detected in the asynchronous host data stream.            */
/************************************************************************/

send_aid  script (int key, char(*) strl, char(*) str2)
          int i

          aid(key)
          wait (60, strl, str2, "LOGIN")

          if (sysret = 3)          /* line drop
            then
              sysret = -99
          endif
ends
/************************************************************************/
```

```
/******************** ERR_MSG SCRIPT ***********************************/
/* The purpose of this script is to determine whether the async host    */
/* connection failed due to a line drop or a time out.                  */
/************************************************************************/

err_msg        script

               if (sysret = -99)
                 then
                   connect (L2)
                   fldpos = $fldaddr(addcust.status)
                   chgattr (L2, fldpos, (U,*,H,*,R,*,*))
                   addcust.status = ("Host Connection Failed.  " +
                                     "Line Dropped.")
                   chgattr (L2, fldpos, (P,*,H,*,R,*,*))
                 else
                   if (sysret = -1)
                     then
                       connect (L2)
                       fldpos = $fldaddr(addcust.status)
                       chgattr (L2, fldpos, (U,*,H,*,R,*,*))
                       addcust.status = ("Host Connection Failed.  " +
                                         "Timed Out.")
                       chgattr (L2, fldpos, (P,*,H,*,R,*,*))
                   endif
               endif
ends
/*************************************************************************/
```

```
        copy "/login.s"
/****************************************************************************/
endp
```

# login.s Subroutine

```
/***********************************************************************
/*                                                                     *
/*                         LOGIN.S                                     *
/*                                                                     *
/***********************************************************************
/*                                                                     *
/*   FUNCTIONAL DESCRIPTION:                                           *
/*      log in to async host                                           *
/*   INPUT PARAMETERS:                                                 *
/*      userid    -> user id                                           *
/*      userpwd   -> user user password                                *
/*   OUTPUT PARAMETERS:                                                *
/*      rtncode  -> 0 = successful log in                              *
/*                  1 = log in rejected                                *
/*                  2 = system not available                           *
/*                                                                     *
/***********************************************************************

login    script (char(*) usid,
                 char(*) uspwd,
                 int rtncode)
         enter
         wait (30,"LOGIN:")
         if (sysret != 1)
           then
             rtncode = 2
             return
         endif
         text (usid)
         enter
         wait (30,"PASSWORD:")
         if (sysret != 1)
           then
             rtncode = 2
             return
         endif
         text (uspwd)
         enter
         wait (30,"ENTER MENU OPTION","LOGIN INCORRECT")
         if (sysret = 1)
           then
             rtncode = 0
           else
             rtncode = 1
         endif
ends
```

## addcust.l Local Screen Format File

```
begfmt addcust
       field (1,30,20,(P,A,H,R,R,7,0)) DUMMY "CUSTOMER ADD SCREEN"
       field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "Service Branch: "
       field (3,22,8, (U,A,H,R,R,7,0)) addcust.branch
       field (5,5,9,  (P,A,H,R,N,7,0)) DUMMY "Name:    "
       field (5,15,35,(U,A,H,R,R,7,0)) addcust.name
       field (7,5,9,  (P,A,H,R,N,7,0)) DUMMY "Street:  "
       field (7,15,35,(U,A,H,R,R,7,0)) addcust.street
       field (9,5,7,  (P,A,H,R,N,7,0)) DUMMY "City: "
       field (9,15,15,(U,A,H,R,R,7,0)) addcust.city
       field (9,32,7, (P,A,H,R,N,7,0)) DUMMY "State: "
       field (9,40,2, (U,A,H,R,R,7,0)) addcust.state
       field (9,44,10,(P,A,H,R,N,7,0)) DUMMY "Zip Code: "
       field (9,55,5, (U,A,H,R,R,7,0)) addcust.zip
       field (11,5,12,(P,A,H,R,N,7,0)) DUMMY "Phone No.: "
       field (11,18,3,(U,A,H,R,R,7,0)) addcust.areacd
       field (11,23,3,(U,A,H,R,R,7,0)) addcust.nnx
       field (11,28,4,(U,A,H,R,R,7,0)) addcust.exch
       field (15,5,21,(P,A,H,R,N,7,0)) DUMMY " Press F8 to EXIT. "
       field (24,2,70,(P,A,H,R,N,7,0)) addcust.status
endfmt
```

# login.l Local Screen Format File

```
begfmt login
      field (1,28,23,(P,A,H,R,R,7,0)) DUMMY "HOST LOGIN SCREEN"
      field (3,5,16, (P,A,H,R,N,7,0)) DUMMY "User Id:    "
      field (3,22,8, (U,A,H,R,R,7,0)) login.userid
      field (5,5,16,(P,A,H,R,N,7,0))  DUMMY "Password:   "
      field (5,22,8,(U,A,D,R,N,7,0))  login.userpwd
      field (24,2,70,(P,A,H,R,N,7,0)) login.status
endfmt
```

# custadd.f Host Screen Format File

```
field (04,48,0008)  custadd.clctrbr
field (04,66,0008)  custadd.clsvcbr
field (05,15,0040)  custadd.clname
field (08,10,0030)  custadd.clstr
field (08,51,0009)  custadd.clgeo
field (09,10,0020)  custadd.clcity
field (09,41,0002)  custadd.clstate
field (09,56,0010)  custadd.clzip
field (11,11,0005)  custadd.clarcd
field (11,17,0003)  custadd.clnnx
field (11,21,0004)  custadd.clexch
field (12,34,0004)  custadd.clsic
field (12,59,0001)  custadd.clstat
field (14,17,0003)  custadd.claess1
field (14,21,0002)  custadd.claess2
field (14,24,0004)  custadd.claess3
field (14,72,0003)  custadd.clprice1
field (16,09,0006)  custadd.clcmu
field (16,21,0030)  custadd.cldesc
```

## chkzip.f Host Screen Format File

```
field (04,27,0015)  chkzip.city
field (04,19,0002)  chkzip.state
field (05,07,0005)  chkzip.rzipmi01
field (05,65,0009)  chkzip.rgeoco01
```

# 4 Commands and Functions


**How to Use This Section** 4-1

**Command Directory** 4-3

ABEND 4-8
AID 4-9
ASSIGN (=) 4-11
ATTN 4-16
BEEP 4-17
BEGFMT/ENDFMT 4-18
BREAK 4-20
BTAB 4-21
CALL 4-22
CAPTURE ON/OFF 4-26
CHAR 4-28
CHGATTR 4-30
CHKPT 4-32
CLEAR 4-34
CLOSE 4-35
COLOR 4-36
COMMENT (/*) 4-38
CONNECT 4-39
COPY 4-43
CURSOR 4-45
CYCLE 4-46
DEL 4-47
DISCON 4-48
DUP 4-50
EJECT 4-51
ENDP 4-52
ENDS 4-53
ENTER 4-54
ERASEW 4-55

| | |
|---|---|
| ERIN | 4-57 |
| EROF | 4-58 |
| EXIT | 4-59 |
| FIELD | 4-63 |
| FM | 4-71 |
| FOR | 4-72 |
| FORMAT | 4-75 |
| FRESH | 4-77 |
| GETFMT | 4-79 |
| GOTO | 4-80 |
| HOME | 4-81 |
| IF | 4-82 |
| INS | 4-84 |
| INT | 4-85 |
| LBREAK | 4-86 |
| LOG | 4-87 |
| NL | 4-89 |
| OPEN | 4-90 |
| PAn | 4-92 |
| PFn | 4-93 |
| PRINT | 4-94 |
| PROG | 4-95 |
| PROMPT | 4-98 |
| PUTENV | 4-100 |
| READ | 4-101 |
| RESET | 4-104 |
| RETURN | 4-105 |
| RUN | 4-106 |
| SCRIPT | 4-107 |
| SERINIT | 4-110 |
| SHOW | 4-114 |
| SWITCH | 4-116 |
| SYSREQ | 4-118 |
| TAB | 4-119 |
| TEXT | 4-120 |
| TIMEOUT | 4-122 |
| WAIT | 4-124 |
| WHILE | 4-127 |
| WINDOW | 4-129 |
| WRITE | 4-132 |
| WTO | 4-134 |

**Function Directory** 4-137


$ATTR 4-141
$CHDATE 4-143
$DATE 4-144
$DATES 4-145
$DAY 4-146
$EVAL 4-147
$FLDADDR 4-150
$GETCUR 4-151
$GETENV 4-152
$GETPID 4-153
$GSUBSTR 4-154
$HEX 4-156
$ITOS 4-157
$LENGTH 4-158
$MONTH 4-160
$NEXTFLD 4-161
$RESP 4-163
$SCAN 4-165
$SEC2TIM 4-168
$STOI 4-169
$STRIP 4-170
$TAB 4-171
$TIMDIFF 4-172
$TIME 4-173
$TIM2SEC 4-174
$YEAR 4-175

# How to Use This Section

This reference section contains a complete alphabetical listing of all ESCORT commands and functions as well as a numerical listing of all error messages.

Listings in the command and function directories contain

- the **name** of the command or function
- the **purpose** or definition
- the **format** or syntax
- comments or **remarks** about using the command or function
- an **example** of how to use the command or function.

At the beginning of each directory, the conventions used throughout, are listed.

Tables indicating which commands and functions are effective in each session type are provided. All commands and functions are effective in all three session types, unless otherwise specifically noted in the *remarks* section in the command and function directories.

# Command Directory

This command directory contains a complete alphabetical listing of all ESCORT commands.

## Conventions Used

Most commands have the following format:

**[label:] COMMAND operands**

Optional fields are noted in brackets. The label in the above example is optional.

Braces indicate a choice of operands. In the following example, you must enter either a string expression or the keyword operand SCREEN.

```
[label:]    PRINT    {str_expr}
                     {SCREEN}
```

Commands and keyword operands are printed in capital letters, but may be entered in either capital or lowercase letters.

Multi-word operands are separated by an underscore. In the following example, the operand *exit_code* represents an exit code number:

**[label:] ABEND [(exit_code)]**

Operands are separated by commas, as in the example below:

**[label:] WRITE (nickname, buffer)**

Parentheses must be entered where indicated. In the example above, the entries for *nickname* and *buffer* must be enclosed in parentheses.

String and integer expressions with multiple operands must be enclosed in parentheses.

The text of a string constant must be entered in double

quotation marks, as shown below:

```
WTO    "This is a window."
```

The names of all scripts, files, programs, variables, and labels must be 1 to 8 characters. The first character must be alphabetic.

Many commands in ESCORT permit you to use a label. A label is a name used to branch to a specified statement during execution. Use of a label is optional.

Upon declaration, a string (or each element in a string array) is initialized to a null string. The term *null string* means a string of length zero.

Upon declaration, an integer variable (or each element in an integer array) is initialized to zero.

Most examples listed in this directory are program sections. Many examples use a dot (.) on a line by itself to denote additional code.

All of the examples listed in this directory show only one command, with in some cases a **COMMENT** marker, on each script line. ESCORT is a free-format programming language and therefore you may write more than one command on each script line. You are limited to a maximum number of commands on a line by the capabilities of your editor.

### Warning

If you write more than one command on a script line, each command *must* be separated by either a blank space or a tab. Do not use a delimiter other than a blank space or tab, such as a semi-colon (;), to separate commands otherwise syntax errors may occur.

## Command Summary

In the following table, a bullet ( ● ) indicates the session type, (synchronous, asynchronous or local) in which each command is effective.

| Command | Synchronous Host | Asynchronous Host | Local |
|---|---|---|---|
| ABEND | ● | ● | ● |
| AID | ● | † | |
| ASSIGN (=) | ● | ● | ● |
| ATTN | ● | | |
| BEEP | ● | ● | ● |
| BEGFMT/ENDFMT | | | ● |
| BREAK | ● | ● | ● |
| BTAB | ● | ● | ● |
| CALL | ● | ● | ● |
| CAPTURE | | ● | |
| CHAR | ● | ● | ● |
| CHGATTR | | | ● |
| CHKPT | ● | ● | ● |
| CLEAR | ● | ● | |
| CLOSE | ● | ● | ● |
| COLOR | | | |
| COMMENT | ● | ● | ● |
| CONNECT | ● | ● | ● |
| COPY | ● | ● | ● |
| CURSOR | ● | ● | ● |
| CYCLE | ● | ● | ● |
| DEL | ● | ● | |
| DISCON | ● | ● | |
| DUP | ● | | |

† In the asynchronous environment, AID keys, corresponding to codes 0 to 8 inclusive and code 25, are effective.

| Command | Synchronous Host | Asynchronous Host | Local |
|---|---|---|---|
| EJECT | | | |
| ENDP | ● | ● | ● |
| ENDS | ● | ● | ● |
| ENTER | ● | ● | |
| ERASEW | ● | ● | ● |
| ERIN | ● | | ● |
| EROF | ● | | ● |
| EXIT | ● | ● | ● |
| FIELD | ● | ● | ● |
| FM | ● | | |
| FOR | ● | ● | ● |
| FORMAT | ● | ● | ● |
| FRESH | ● | ● | |
| GETFMT | | | ● |
| GOTO | ● | ● | ● |
| HOME | ● | ● | ● |
| IF | ● | ● | ● |
| INS | ● | | ● |
| INT | ● | ● | ● |
| LBREAK | | ● | |
| LOG | ● | ● | ● |
| NL | ● | ● | ● |
| OPEN | ● | ● | ● |

| Command | Synchronous Host | Asynchronous Host | Local |
|---|---|---|---|
| PAn | ● | | |
| PFn | ● | † | |
| PRINT | ● | ● | ● |
| PROG | ● | ● | ● |
| PROMPT | | ● | |
| PUTENV | ● | ● | ● |
| READ | ● | ● | ● |
| RESET | ● | | |
| RETURN | ● | ● | ● |
| RUN | ● | ● | ● |
| SCRIPT | ● | ● | ● |
| SERINIT | | ● | |
| SHOW | ● | ● | ● |
| SWITCH | ● | ● | ● |
| SYSREQ | ● | | |
| TAB | ● | ● | ● |
| TEXT | ● | ● | ● |
| TIMEOUT | ● | ● | ● |
| WAIT | ● | ● | ● |
| WHILE | ● | ● | ● |
| WINDOW | ● | ● | ● |
| WRITE | ● | ● | ● |
| WTO | ● | ● | ● |

† In the asynchronous environment, keys *PF1* to *PF8* inclusive, are effective.

# ABEND

**Purpose** Terminates execution of ESCORT abnormally and returns an exit code to the UNIX shell.

**Format** [label:] ABEND [(exit_code)]

**exit_code** specifies a code that is sent to the UNIX shell. This exit code can be tested in a UNIX shell script. A zero (normal) exit code is returned if this operand is not specified.

**Remarks** The exit code can be an integer constant or an integer variable with a value between 0 and 255. ESCORT returns a zero (normal) exit code to the UNIX shell when the **ENDP** statement is encountered (the last statement in the program).

## Example

```
WTO  "Failed To Add Order - Program Abend S12"
ABEND (12)    /* terminate with user code 12
```

## AID

**Purpose** Simulates the action of one of the attention-identifier (AID) keys on the keyboard. The AID keys are:

- in the synchronous environment
  ENTER
  PF1 - PF24
  CLEAR
  PA1 - PA3
  ATTN
  SYS_REQ
- in the asynchronous environment
  ENTER
  PF1 - PF8 (corresponding to soft function keys F1 to F8 )
  CLEAR

**Format** **[label:] AID (n)**

**n** specifies the code representing the AID key you want to simulate. The key code can be an integer constant or an integer variable. The following values have been assigned:

| AID key | Code |
|---|---|
| ENTER | 0 |
| PF1 | 1 |
| PF2 | 2 |
| . | . |
| . | . |
| PF24 | 24 |
| CLEAR | 25 |
| PA1 | 26 |
| PA2 | 27 |
| PA3 | 28 |
| ATTN | 29 |
| SYS_REQ | 30 |

**Remarks** This command is effective in synchronous and asynchronous sessions.

After an **AID** command is executed, when connected to an active synchronous host session, script execution is suspended until the keyboard is unlocked.

**See also** **ATTN, CLEAR, CONNECT, ENTER, PAn, PFn**, and **SYSREQ** commands.

**Example** The following example sends PF1 to PF12 to the synchronous host. After each response from the synchronous host system, the PF key number is printed.

```
FOR  i=1 TO 12
DO
    AID (i)     /* send PFi and wait for host response
    PRINT ("PF" + $ITOS(i))
ENDDO
```

# ASSIGN (=)

**Purpose** Assigns a value returned from an expression to a variable. The assignment operation is a data move operation. On the left side of the equal sign is the name of the destination *variable*. It receives data evaluated from the right side *expression* (source data).

**Format**

| *Variable* | | *Expression* | *Assignment Type* |
|---|---|---|---|
| [label:] int_var | = | int_expr | integer variable |
| [label:] int_array | = | (int_const,...) | integer array |
| [label:] int_array(i) | = | int_expr | integer array element |
| [label:] str_var | = | str_expr | string variable |
| [label:] str_array | = | (str_const,...) | string array |
| [label:] str_array(i) | = | str_expr | string array element |
| [label:] scrn_fld | = | str_expr | screen field variable |

**int_var** specifies an integer variable.
**int_expr** specifies an integer expression.
**int_array** specifies an integer array.
**int_const** specifies an integer constant.
**int_array(i)** specifies an integer array element.
**str_var** specifies a string variable.
**str_expr** specifies a string expression.
**str_array** specifies a string array.
**str_const** specifies a string constant.
**str_array(i)** specifies a string array element.
**scrn_fld** specifies a screen field variable.

**Remarks**

A string expression may contain a string constant, string variable, string array element, string function, screen field variable, or more than one of the above operands separated by the concatenation operator (+ sign).

An integer expression may contain an integer constant, integer variable, integer array element, integer function or more than one of the above operands separated by an integer operator.

A relational expression, when evaluated, always returns an integer value. A zero value yields a false condition and a non-zero value yields a true value. A relational expression is also considered an integer expression.

If multiple operands are used in either an integer expression or in a string expression, then the entire expression must be enclosed in left and right parentheses.

A string constant containing a character string must be enclosed in double quotes.

A variable must be declared before it can be used in an assignment statement. Variables are declared by using **INT, CHAR,** or **FIELD** statements. The scope of a variable may be *local* or *global.*

If the length of the right side (source data) in a string or screen field assignment statement is more than the length of the left side (destination field), then the assignment terminates when the destination field is full. An overflow condition is *not indicated* by ESCORT.

For example,

CHAR (8) lastname

.

.

lastname = "Frankenberger"

moves the first 8 characters, *Frankenb*, to the string variable *lastname*. The remaining characters, *erger*, are lost, but no error is reported.

**See also** **CHAR, FIELD,** and **INT** commands, and the section , "Operators and Expressions", in Chapter 2.

**Example 1** The following example demonstrates various types of integer variable assignments:

```
/*    Declarations
INT   i                  /* integer
INT   j(6)               /* integer array
INT   k                  /* integer
/*    Assignments
k  =  236                      /* integer constant
i  =  k                        /* integer variable
i  =  j(2)                     /* integer array element
i  =  $GETCUR                  /* integer function
i  =  (($GETCUR/236)+j(k))     /* multiple operands
```

**Example 2** The following example demonstrates integer array initialization:

```
/*   Declaration
INT  j(6)                          /* integer array
/*   Assignment
j = (256,0,-1,32,32767,-32767)     /* integer array initialization
```

**Example 3** The following example demonstrates integer array element assignments:

```
/*    Declarations
INT   j(6)                 /* integer array
INT   k                    /* integer
/*    Assignments
k     =    4
j(3)  =   -1                /* integer constant
j(k)  =    k                /* integer variable
j(k)  =    j(3)             /* integer array element
j(5)  =    $GETCUR          /* integer function
j(3)  =    ($GETCUR-1)      /* multiple operands
```

**Example 4** The following example demonstrates string variable assignments:

```
/*    Declarations
CHAR   (20)  u                 /* string
CHAR   (15)  v                 /* string
CHAR   (10)  y (3)             /* string array
FIELD  (2,12,15) fl            /* screen field

/*    Assignments
u  =  "$ 1,800.00"              /* string constant
v  =  u                         /* string variable
u  =  y(2)                      /* string array element
v  =  $DATE                     /* string function
u  =  fl                        /* screen field variable
v  =  ("DATE = " + $DATE)       /* multiple operands
```

**Example 5** The following example demonstrates string array initialization:

```
/*   Declaration
CHAR (10) y(3)                      /* string array

/*   Assignments
y =   ("cereal","sugar","milk")  /* string array initialization
```

**Example 6** The following example demonstrates string array element assignment:

```
/*    Declarations
CHAR   (20)  u                 /* string
CHAR   (15)  v                 /* string
CHAR   (15)  y (3)             /* string array
FIELD  (2,12,15) fl            /* screen field
INT     k                      /* integer

/*    Assignments
k      =   2
v      =   $DATE
y(1)   =   "sugar"              /* string constant
y(k)   =   v                    /* string variable
y(2)   =   y(1)                 /* string array element
y(k)   =   $DATE                /* string function
y(3)   =   fl                   /* screen field variable
y(1)   =   ("TIME = " + $TIME)/* multiple operands
```

**Example 7** The following example demonstrates screen field variable assignments:

```
/*  Declarations
FIELD  (10,5,20)  f1        /* screen field
FIELD  (15,10,15) f2        /* screen field
CHAR   (15)  y(3)           /* string array
CHAR   (20) u               /* string

/*  Assignments
u   =  "123.25"
f2  =  "hello"               /* string constant
f1  =  u                     /* string variable
f2  =  y(3)                  /* string array element
f1  =  $DATE                 /* string function
f1  =  f2                    /* screen field variable
f2  =  ("DATE = " + $DATE)   /* multiple operands
```

**Example 8** This example demonstrates a *special case* of the assignment statement:

You can use a *special case* of the assignment statement to initialize a string variable with a pattern. For example:

$x = (y+x)$, where $x$ and $y$ are strings, is equivalent to $x = (y+y+y+...)$.

In this example, the pattern $y$ is propagated throughout $x$. Propagation will be repeated according to the declared size of string $x$.

You can also use the special assignment statement to propagate blanks or dashes throughout a field. For example:

```
CHAR (10) S
  .
  .
S = ("-" + S)    /* s = "----------"
S = (" " + S)    /* s = "          "
```

# ATTN

**Purpose** Simulates the action of the attention key on the keyboard.

**Format** [label:] ATTN

**Remarks** This command is effective in synchronous sessions.

After an **ATTN** command is executed, script execution is suspended until the keyboard is unlocked.

This key is used by certain applications in an SNA/SDLC environment.

**Example**

```
ATTN                /* interrupt program execution.
```

# BEEP

**Purpose** Sounds a beep on your terminal to alert you to a particular condition.

**Format** [label:] BEEP

**Example** The following example uses the **BEEP** command to beep 3 times before entering interactive mode.

```
WTO     "PRESS PA2 TWICE, THEN PRESS F2"
FOR     i=1 TO 3      /* sounds 3 beeps
DO
        BEEP
ENDO
EXIT                  /* enter interactive mode
```

## BEGFMT/ENDFMT

**Purpose**

Marks the beginning and end of local screen format definition.

**Format**

BEGFMT **screen_name**

•

•

ENDFMT

**screen_name** specifies the local screen format name. The *screen_name* consists of from one to eight alphanumeric characters, the first character of which must be alphabetic. Individual *screen_names* must be unique within a script.

**Remarks**

This command is effective in local sessions.

**BEGFMT/ENDFMT** are administrative commands. Up to 100 local screen formats can be defined in a single script, each of which may contain a maximum of 500 fields.

Local screen format definitions must be written after the **PROG** statement and before the first **SCRIPT** statement.

**FIELD** statements are written between the **BEGFMT** and **ENDFMT** statements. An unformatted screen containing a single unprotected field of 1920 characters will be created by using a **BEGFMT** and an **ENDFMT** statement without an intervening **FIELD** statement.

**See also**

**FIELD** statement and **FORMAT** command.

**Example** In this example two local screen formats, the *order* format and the *logon* format are created.

```
progl   PROG    main
        .
        .
        BEGFMT  order
          FIELD (1,2,9,(P,*,H,*,*,*,*)) DUMMY "ORDER # :"
          FIELD (1,12,8,(*,N,*,*,R,*,*)) ordno
          .
          .
        ENDFMT
        BEGFMT  logon
          FIELD (10,12,9,(P,*,H,*,*,*,*)) DUMMY "PASSWORD:"
          FIELD (10,22,8,(*,*,D,*,*,*,*)) passwd
          .
          .
        ENDFMT
main    SCRIPT
        .
        .
```

# BREAK

**Purpose** Discontinues processing of a loop within your program.

**Format** [label:] BREAK

**Remarks** The **BREAK** command is used to break from a **WHILE** or **FOR** loop. When used between **DO** and **ENDO**, it causes a branch to the statement following **ENDO**.

**See also** **CYCLE**, **FOR**, and **WHILE** commands.

**Example** This program calls a subroutine, ADDORDER, in a loop. The subroutine returns a code. The program checks the code and terminates the loop if a code other than zero is returned.

```
FOR i = 1 to 20
DO
    CALL  ADDORDER
    IF    code != 0
    THEN
          PRINT ("FAILING CODE + " $ITOS(code))
          BREAK                /* quit loop
    ENDIF
ENDO
```

# BTAB

**Purpose** Simulates action of the back-tab key on the keyboard.

**Format** [label:] **BTAB** [(n)]

**n** specifies the number of back-tabs to be performed. The *n* can be an integer constant or an integer variable. It can have a value between 1 and 64. The default value for *n* is 1.

**See also** **TAB** command and **$TAB** function.

**Example** This example demonstrates use of the **BTAB** command to find the first unprotected field before the literal "ORDER#".

```
K = $SCAN ("ORDER#", (12,1,100))
CURSOR (K)                  /* position cursor at literal
BTAB                        /* backup to the first unprotected field
                            /* before literal "ORDER#"
```

# CALL

**Purpose** Invokes another script.

**Format** [label:] CALL script_name [(parm_list)]

**script_name** specifies the name of the script to be executed.

**parm_list** specifies the list of parameters to be passed to or returned from a script. The *parm_list* is optional and may contain integer constants, integer variables, string constants, string variables or field variables. Note that arrays, array elements, and functions are *not* allowed in the *parm_list*. If you are specifying a parameter list, you must enclose it in parentheses.

For each parameter in the *parm_list* in the **CALL** statement, there must be a corresponding entry in the *decl_list* in the **SCRIPT** statement. Each type of parameter in the *parm_list* and *decl_list* must be consistent. See the table below for examples.

| parm_list | decl_list |
|---|---|
| integer constant | integer variable |
| integer variable | integer variable |
| string constant | string variable |
| string variable | string variable |
| field variable | screen field variable or string variable |

**Remarks** The **CALL** command is similar to the subroutine call in other programming languages.

You may nest calls. For example, if script A calls script B, script B may contain a call to script C.

The variable names used in the *parm_list* may be the same as in the *decl_list*. You may not use arrays, array elements or functions in the parameter list.

The length of each passed variable is assigned to its corresponding local variable in the *decl_list* when a subroutine is executed. Therefore, the length of a local variable is not explicitly declared in the *decl_list* but is marked by an asterisk instead. Further details on passing variables are provided in the **SCRIPT** statement.

Called scripts may be defined internally within the same program as the **CALL** command, or externally in a separate file. If the called scripts are defined externally, they must be included in the calling program by use of the **COPY** command.

An ESCORT script is a procedure and not a function. To return a value from an ESCORT script, you must pass a suitable parameter in the *parm_list*.

**See also** **COPY** command and **SCRIPT** statement.

**Example 1** The first example shows global variables used as parameters. Script *s1* calls script *s2*. Return from *s2* is made via a **RETURN** or **ENDS**. The subroutine returns a value which is assigned to *orderno*.

```
p1    PROG  s1              /* start of program p1
       .
      CHAR (10) custid      /* global variables
      CHAR (6)  reqdue
      CHAR (9)  orderno
      FIELD (12,23,9) cust.order
s1    SCRIPT
       .
       .    (more code)
       .
      custid = "000000414" /*globals used in s2
      reqdue = "073184"
      CALL  s2 (orderno)    /* call script s2
```

```
    .
    .  (more code)
    .
    ENDS                  /* end of script sl
s2  SCRIPT (char (*) ordparm)
    CHAR (10) a           /* local variables
    CHAR (6)  b
    .
    .  (more code)
    .
    ordparm = cust.order /* return parameter (orderno)
    IF      (custid = a) & (reqdue = b)
    THEN
            RETURN        /* return to sl
    ENDIF
    .
    .  (more code)
    .
    ENDS                  /* return to sl
    ENDP                  /* end of program
```

**Example 2** The next example is the same as the previous example except that local variables are used to pass and return parameters. Note that the first parameter, *custid*, is used as a string constant in the call.

```
p2  PROG  sl                       /* start of program p2
    FIELD (12,23,9) cust.order
sl  SCRIPT                         /* start of script sl
    CHAR (10) custid               /* local variables
    CHAR (6)  reqdue
    CHAR (9)  orderno
    .
    .    (more code)
    .
    reqdue = "073184"
    CALL  s2 ("000000414", reqdue, orderno)
    .
    .  (more code)
    .
    ENDS                           /* end of script sl
s2  SCRIPT (CHAR (*) customer,     /* input parm, string constant
            CHAR (*) duedat,       /* input parm, string variable
            CHAR (*) ordparm)      /* output parm, string variable
    CHAR (10) a                    /* local variables
    CHAR (6)  b
    .
    .  (more code)
    .
    ordparm = cust.order            /* return parameter (orderno)
    IF      (customer = a) & (duedat = b)
    THEN
             RETURN                /* return to sl
    ENDIF
    .
    .  (more code)
```

```
     .
     ENDS                          /* return to sl
     ENDP                          /* end of program
```

**Example 3** In the last example, four parameters are passed. Two will contain returned values.

```
progl PROG    main
main  SCRIPT
      INT        code                  /* output parm - integer
      CHAR (80) response               /* output parm - string
      CALL       sub  (80, code, "ADD COMPLETED", response)
      IF         code = 0
      THEN       PRINT "SUCCESSFUL ADD"
      ELSE       PRINT "ADD FAILED"
      ENDIF
      PRINT      response
      ENDS
sub   SCRIPT     (INT  length,         /* integer constant - input
                  INT  rtncode,        /* integer variable - output
                  CHAR (*) message,    /* string constant - input
                  CHAR (*) response,)  /* string variable - output
      FIELD (24,1,20) line24
      PF4
      IF         $SCAN (message, (24,1,length))
      THEN       rtncode = 0
      ELSE       rtncode = -1
      ENDIF
      response = line24
      ENDS
      ENDP
```

## CAPTURE ON/OFF

**Purpose** Toggles on and off the capture of output from an asynchronous host.

**Format** [label:] CAPTURE ON

.

.

[label:] CAPTURE OFF

**Remarks** This command is effective in asynchronous sessions.

The capture feature may be turned on and off as necessary during script execution. Each time **CAPTURE** is turned on, all data received from the asynchronous host is captured and is appended to the file named *escort.cp{proc-id}*, where *{proc-id}* refers to the unique process identification the UNIX operating system assigns to each process. The file is created in the directory defined by the ESCDIR environment variable.

Checking for a successful **CAPTURE ON** operation, when the *escort.cp{proc-id}* file is first created, is good programming practice. The internal global integer variable, *SYSRET*, returns the result of a **CAPTURE ON** operation. *SYSRET* may have one of the following values after the **CAPTURE ON** is executed:

| | |
|---|---|
| 0 | Successful **CAPTURE ON** |
| non-0 | Failed **CAPTURE ON** |

The command will fail if the output file, *escort.cp{proc-id}* cannot be created. A message will be written to the *escort.pr{proc-id}* file.

**Example** In this example, asynchronous host system responses are captured.

.

.

```
CAPTURE ON
TEXT "Johnson, J."
ENTER                        /* send information to host
WAIT (10, "Add Complete")    /* wait for host system response
.
.
CAPTURE OFF
.
.
```

# CHAR

**Purpose** Declares a string variable or a string array.

**Format**

**CHAR** **(size) name**
*or*
**CHAR** **(size) name (#elements)**

**size** specifies the maximum size of a character string or an array element. The actual size changes each time a string is assigned. The *size* may be between 1 and 2048, inclusive.

**name** specifies the name of the variable. The *name* may be between 1 and 8 characters. The variable name must not be a reserved word. Further details on naming variables may be found in the section, "Naming Variables", in Chapter 2.

**#elements** specifies the number of elements in an array. The array may contain 1 to 2048 elements, inclusive. Further details on array elements may be found in the sections, "String Variables", and "String Array Variables" in Chapter 2.

**Remarks** Upon declaration, the string (or each element in the array) is initialized to a null value and has a zero length.

A string (or a string element) may be assigned a string expression or a screen field variable by using an assignment statement.

## Example

```
CHAR (20)name              /* string variable
CHAR (9) orderno           /* string variable
CHAR (2) table2 (5)        /* string array
   .
name  =  "JOHNSON, L.B"    /* string assignment
   .
table2 = ("ab", "c", "e", "GH", "15")  /* array initialization
   .
name = "DAVIS Jr., S."     /* string reassignment
   .
table2 (1) = "cd"          /* string element reassignment
```

# CHGATTR

**Purpose** Changes the field attributes for a given local screen format.

**Format** [label:] CHGATTR (local_session-id, position, (attr_list))

**local_session-id** specifies a local session identification. Valid local session identifications are:

| | |
|---|---|
| L1 | Local session 1 |
| L2 | Local session 2 |

**position** specifies the absolute screen address of the first position of the field for which the attributes are to be changed. The *position* can be an integer constant or an integer variable within the range from 1 to 1920.

**attr_list** specifies the new attribute list to be applied to the field for which the attributes are to be changed. The *attr_list* follows the same format as the attribute list in the **FIELD** statement. You may use an asterisk (*) to specify the default attribute in any of the seven attribute groups.

**Remarks** This command is effective in local sessions.

If the absolute address of the starting position of the field, for which the attributes are to be changed, is unknown, it can be obtained by using either a **$FLDADDR** or a **$GETCUR** function.

**See also** **FIELD** statement and **$FLDADDR** and **$GETCUR** functions.

**Example** In this example, the field named *order#* is defined in the local screen format section as follows:

```
FIELD (1,12,8,(*,N,*,*,*,*,*)) order#
```

The field is defined as numeric with all other attributes using the default values. If the user enters an incorrect order number, the script will prompt the user to key-in the correct order number and will change the attributes of the *order#* field to underline to highlight the error.

```
INT i
.
.
i=$FLDADDR(order#)                  /* get field address
CHGATTR (L1, i, (*,N,*,*,U,*,*))    /* underline
WINDOW (21,15,24,35)                /* window for message
WTO "Incorrect Order #"
WTO "Correct - press ENTER"
BEEP                                /* to draw attention
EXIT                                /* go to interactive mode
.
.
```

## CHKPT

**Purpose** Preserves the contents of an active file in case of a system disaster.

**Format**

```
[label:]    CHKPT    ({nickname})
                      {LOG}
                      {PRN}
                      {CAP}
```

**nickname** specifies the internal name of the file. This name must have been previously defined in an **OPEN** command. The *nickname* is global and can be used in any script within the program.

**LOG** specifies the ESCORT log file, which is named *escort.lg{proc-id}*.

**PRN** specifies the ESCORT print file, which is named *escort.pr{proc-id}*.

**CAP** specifies the ESCORT capture file, which is named *escort.cp{proc-id}*.

*{proc-id}* is the unique process identification that the UNIX operating system assigns to the particular process.

The files are created in the directory defined by the ESCDIR environment variable.

**Remarks** Data is not written from the internal system buffer to the file unless the internal system buffer is full or a **CHKPT** command in a script is encountered. In the event of a system failure, data in the internal system buffer is lost. If data is critical, therefore, a **CHKPT** command should be performed after each **WRITE** command. Such frequent use of the **CHKPT** command may cause slight degradation in script performance.

**See also** **OPEN** and **WRITE** commands.

**Example** This example saves the contents of a file after every 20 records.

```
OPEN  (file1, "file1.f", W)
k = 1
WHILE (k < 100)
  DO
    FOR i = 1 to 20
      DO
        buffer = rec (k)       /* get next record
        k = (k + 1)
        WRITE  (file1, buffer) /* write it
      ENDO
    CHKPT (file1)              /* checkpoint every 20 records
  ENDO
```

# CLEAR

**Purpose** Simulates the action of the clear key on the keyboard.

**Format** [label:] CLEAR

**Remarks** This command is effective in synchronous and asynchronous sessions.

When connected to an active synchronous host session, after a **CLEAR** command is executed, script execution is suspended until the keyboard is unlocked.

**See also** **AID** and **CONNECT** commands.

## Example

```
TEXT "Add information to screen"
ENTER                        /* send information to host
CLEAR                        /* clear screen
TEXT "/for mainmenu"
ENTER                        /* go to main menu
```

# CLOSE

**Purpose** Closes a file.

**Format** [label:] CLOSE **(nickname)**

**nickname** specifies the internal name for the file. This must be the same name assigned to the file in the **OPEN** command. The *nickname* is global and can be used in any script within the entire program.

**Remarks** Checking for a successful **CLOSE** operation is good programming practice. The internal global integer variable, *SYSRET*, returns the result of a close operation. *SYSRET* may have one of the following values after the **CLOSE** is executed:

| | |
|---|---|
| 0 | Successful **CLOSE** |
| −1 | Failed **CLOSE** |

All files are closed automatically by ESCORT at the end of program execution.

**See also** **OPEN** command.

**Example** In this example a file, F, is closed. A status check using *SYSRET* is made after the file is closed. If the **CLOSE** failed (SYSRET= −1), a message is written to the *escort.pr{proc-id}* file and execution continues.

```
CLOSE  (F)              /* close file
IF     SYSRET < 0        /* check value of SYSRET
THEN
       PRINT ("FAILED TO CLOSE FILE")
       BEEP              /* sound alarm to alert operator and continue
ENDIF
```

# COLOR

**Purpose** Specifies colors used in creating a window.

**Format** [label:] COLOR (frame, background, foreground)

**frame** is a numeric code that defines the color of the window borders. The table below lists the possible colors and codes you may use.

**background** is a numeric code that defines the color of the window background. You may use only the colors listed in Column 1 of the table below for the background of a window.

**foreground** is a numeric code that defines the color of the window foreground. The table below lists the possible colors and codes you may use.

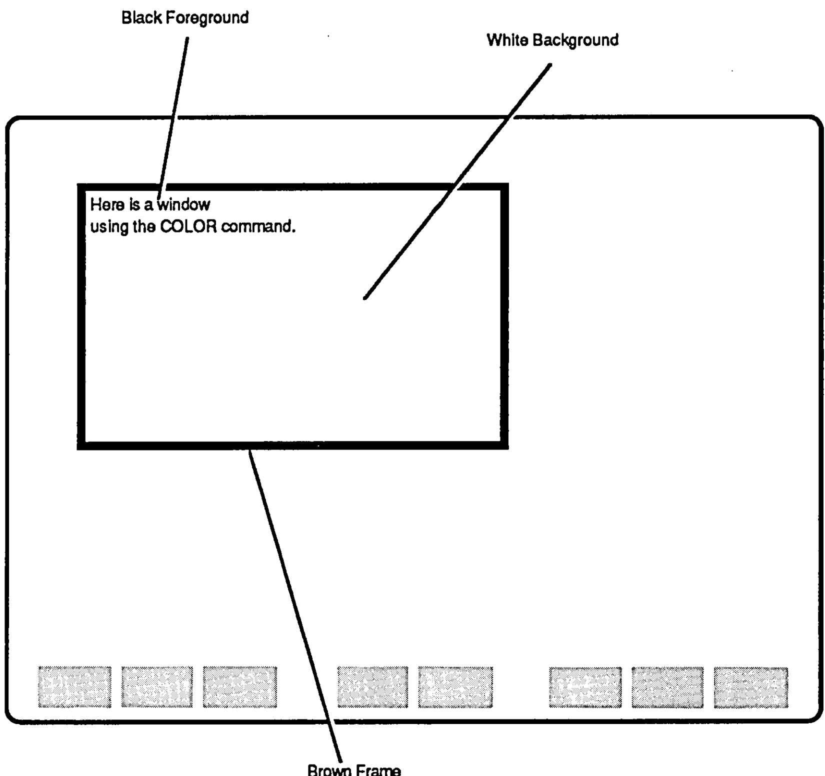
| Code | Color | Code | Color |
|---|---|---|---|
| 0 | Black | 8 | Gray |
| 1 | Blue | 9 | Light Blue |
| 2 | Green | 10 | Light Green |
| 3 | Cyan | 11 | Light Cyan |
| 4 | Red | 12 | Light Red |
| 5 | Magenta | 13 | Light Magenta |
| 6 | Brown | 14 | Yellow |
| 7 | White | 15 | High Intensity White |

**Remarks** The **COLOR** command applies to the MS-DOS version of ESCORT and is not available in the UNIX operating system version of ESCORT. It is included for script compatibility between the UNIX operating system version and MS-DOS operating system version of ESCORT.

**See also** **WINDOW** command.

## Example

```
COLOR (6,7,0)      /* brown frame, white background, black foreground
WINDOW (3,5,15,50) /* draw window
WTO "Here is a window"
WTO "using the COLOR command."
```



# COMMENT (/*)

**Purpose** Indicates the beginning of comments on a line.

**Format** **/*.....This is a comment.....**

**Remarks** A comment may be placed anywhere on a statement line or on a line by itself.

The beginning of a comment is marked with a slash (/) and an asterisk (*). A comment is terminated at the end of the line. If you want the comment to exceed one line, start each continuation line with a "/*".

You may use upper and lower-case characters, numbers, or special characters in a comment.

You may include as many comments in your program as you like.

## Example

```
PF1             /* comment after command
/******************entire line of comments******
TEXT ("abcdefghijklmnop" + /* comment in middle
      "qrstuvwxyz")
```

# CONNECT

**Purpose**

Opens and makes active a particular session, or if the defined session is already open, activates that session.

**Note**

The value of the system global variable, *SYSRET*, must be checked to ensure script integrity.

**Format**

**[label:] CONNECT (session-id)**

**session-id** specifies a session identification. Valid session identifications are:

| | |
|---|---|
| H1 | Synchronous host session 1 |
| H2 | Synchronous host session 2 |
| H3 | Synchronous host session 3 |
| H4 | Synchronous host session 4 |
| A1 | Asynchronous host session 1 |
| A2 | Asynchronous host session 2 |
| A3 | Asynchronous host session 3 |
| A4 | Asynchronous host session 4 |
| L1 | Local session 1 |
| L2 | Local session 2 |

**Remarks**

Check for a successful **CONNECT** operation when the connection to the host is first established. The internal global integer variable, *SYSRET*, returns the result of a connect operation. *SYSRET* may have one of the following values after the **CONNECT** is executed:

| | |
|---|---|
| 0 | Successful **CONNECT** |
| non-0 | Failed **CONNECT** |

The **CONNECT** command defines a session as active by making the associated presentation space active. Only one session can be active at any given time.

When an ESCORT script is started, the synchronous host session, H1, is the active session by default. This default can be changed by specifying another session identification in the **PROG** statement.

If an asynchronous session is specified by the **CONNECT** *session-id*, ESCORT physically connects to the asynchronous host using the communication port initialization parameters specified by an associated **SERINIT** statement.

Data can be manipulated in an active session's presentation space. Refer to the tables preceding the command and function directories to determine which commands and functions are effective in the synchronous, asynchronous and local environments.

ESCORT handles **AID** commands and AID keys in a special manner depending upon whether a host or local session is made active by use of the **CONNECT** command (or by setting the *session-id* operand in the **PROG** statement).

- When a synchronous host session is active:
  - In *Script* mode, all **AID** commands are sent to the synchronous host.
  - In *Interactive* mode, all AID keys are sent to the synchronous host.
  - In *Tutorial* mode, none of the AID keys is sent to the synchronous host. The script is resumed and the value of the AID key entered is available by accessing the system global variable, *SYSAID*.
- When an asynchronous host session is active:
  - In *Script* mode, only the **ENTER, PF1** to **PF8** (corresponding to soft function keys F1 to F8 ) and **CLEAR AID** commands are sent to the asynchronous host, all other **AID** commands are ignored.

- In *Interactive* mode, only the ENTER, PF1 to PF8 (corresponding to soft function keys (F1) to (F8) ) and CLEAR keys are sent to the asynchronous host, all other AID keys are ignored.
- In *Tutorial* mode, none of the AID keys is sent to the asynchronous host. The script is resumed and the value of the AID key entered is available by accessing the system global variable, *SYSAID*.

□ When a local session is active:

- In *Script* mode, all **AID** commands are ignored.
- In *Tutorial* mode, none of the AID keys is sent to the host. The script is resumed and the value of the AID key entered is available by accessing the system global variable, *SYSAID*.

**See also** **AID, DISCON, PROG**, and **SERINIT** commands, and the section, "Asynchronous Communication Port Initialization", in Chapter 2.

**Example** In the following example, an order number is entered into a local screen format. The script uses the **CONNECT** command to make the synchronous host session, H1, active. The order number is displayed in the correct synchronous host field position.

```
.
.
CHAR  (10)  order                         /* string variable to transfer data
.
.
BEGFMT stock                              /* local screen format
 FIELD (5,8,10,(*,*,*,*,*,*,*)) l_order   /* local session field variable
ENDFMT
.
.
FIELD (9,25,10) h_order                   /* host session field variable
.
.
GETFMT (L1, stock)                        /* load local screen format
```

```
CONNECT (L1)                          /* connect to local session
SHOW (L1)                             /* display local screen format
EXIT                                  /* go to tutorial for data
order=l_order                          /* data to transfer variable
CONNECT (H1)                          /* connect to host session
IF (SYSRET = -1)                       /* check connection
  THEN
  PRINT ("Connection to Host Failed.")
  EXIT
ENDIF
SHOW (H1)                             /* display host session
h_order=order                          /* order number automatically
                                      /* entered in correct host
                                      /* screen position
```

# COPY

**Purpose**

Includes a specified file in your source program.

**Format**

**COPY "filename"**

**filename** specifies the file you want to include.

**Remarks**

The **COPY** command is a preprocessor command that alters your source code by including a file that you specify. The copied file appears in your program beginning at the location of the **COPY** command.

Up to 100 characters are allowed in a *filename*. You may use either the filename or the complete pathname of a file.

You may nest a **COPY** command within a copied file. However, only two-level nesting is permitted. You will receive an error message if you attempt three-level nesting.

The **COPY** command can be coded anywhere between the **PROG** and **ENDP** statements. It is recommended, however, that you copy all the subroutine scripts right before the **ENDP** statement. Global variables should be copied immediately after the **PROG** statement and local variables should be copied immediately after the **SCRIPT** statement in the appropriate scripts.

Scripts are portable between the UNIX operating system version and the MS-DOS operating system version of ESCORT and you may, therefore, substitute the standard UNIX operating system slash character (/) in a UNIX file pathname with the MS-DOS back-slash (\) file name separation character when using the **COPY** command.

**Example** In this example, use of the **COPY** command copies the file named *myfile* into the program from the */usr/myname* directory.

```
COPY   "/usr/myname/myfile"
```

# CURSOR

**Purpose** Positions the cursor at a specified location on the screen.

**Format** [label:] CURSOR {(row,col)}
{(position)}

**row, col** specifies a desired cursor address in row and column numbers. The *row* and *col* may be either integer constants or integer variables.

**position** specifies a desired cursor address in the form of screen offset +1. For example, the first position on the screen is 1 and the last position is 1920. The *position* may be either an integer constant or an integer variable.

**Example**

```
INT     i
  .
  .
CURSOR  (6,10)            /* positions cursor on row 6, col. 10
TEXT    "Cursor is here"  /* writes data at row 6, col. 10
  .
  .
FOR   i=1 to 24
DO
      CURSOR (i,1)        /* position cursor at row i, col. 1
      TEXT "Hello"        /* writes "Hello" at specified cursor location
      ENTER
ENDO
  .
  .
CURSOR  (1155)            /* absolute-screen position
TEXT    "ABSOLUTE POSITION"
```

# CYCLE

**Purpose** Skips to the next iteration of a loop in a **WHILE** or **FOR** statement.

**Format** [label:] CYCLE

**Remarks** The **CYCLE** command is complementary to the **BREAK** command. The **CYCLE** command branches processing of the program to the next repetition of the loop. The **BREAK** command can be used to branch outside the loop.

**See also** **BREAK, FOR,** and **WHILE** commands.

**Example** This example shows use of both the **CYCLE** and **BREAK** commands. An array of names is printed. The name *MILLER* will not be printed. If the name *JOHNSON* is encountered, the printing process terminates.

```
CHAR (20) table (6)        /* declares 6 entries in a table
CHAR (20) name             /* declares a name string
INT  i                     /* declares a table entry number

table = ("BROWN",           /* initializes table
         "JONES",
         "SMITH",
         "MILLER",
         "WHITE"
         "JOHNSON")

FOR  i = 1 to 6
DO
   name=table(i)            /* get name from table
   IF   name="MILLER"
   THEN
        CYCLE              /* skip PRINT
   ENDIF
   IF   name="JOHNSON"
   THEN
        BREAK              /* stop executing loop
   ENDIF
   PRINT name              /* print name
ENDO
```

## DEL

**Purpose**

Simulates the action of the delete key on the keyboard.

**Format**

[label:] DEL [(n)]

**n** specifies the number of times you want to repeat execution of the delete key. The *n* can be an integer constant or integer variable and have a value of 1 to 64, inclusive. The default value for *n* is 1.

**Remarks**

This command is effective in synchronous and asynchronous sessions.

**Example**

```
                    /* field at row 5, col 11 contains
                    /* incorrect date (auAugust 30, 1985)
CURSOR (5,11)       /* position cursor on field containing
                    /* wrong information
DEL (2)             /* delete first 2 characters of field
```

## DISCON

**Purpose** Closes a particular host session.

**Format** [label:] DISCON (host_session-id)

**host_session-id** specifies a host session identification. Valid host session identifications are:

| | |
|---|---|
| H1 | Synchronous host session 1 |
| H2 | Synchronous host session 2 |
| H3 | Synchronous host session 3 |
| H4 | Synchronous host session 4 |
| A1 | Asynchronous host session 1 |
| A2 | Asynchronous host session 2 |
| A3 | Asynchronous host session 3 |
| A4 | Asynchronous host session 4 |

**Remarks** This command is effective in synchronous and asynchronous sessions.

The **DISCON** command terminates the specified host session and releases the host system connection. Use of **DISCON** does not log you off from a host application; you should follow the usual logoff procedure before using the **DISCON** command in a script.

If the specified host session is the currently connected, active session, the **DISCON** command will, after terminating the specified host session, connect the ESCORT script to the lowest available host session, if any, within the same environment. If no other host session is available in the same environment, the ESCORT script is connected to the lowest available host session.

If the specified host session is a dormant session, the **DISCON** command will terminate the specified host session; the currently connected, active session is not affected.

If only one host session is connected when the ESCORT script encounters the **DISCON** command, ESCORT is automatically connected to local session L1.

Checking for a successful **DISCON** operation is good programming practice. The internal global integer variable, *SYSRET*, returns the result of a disconnect operation. *SYSRET* may have one of the following values after the **DISCON** is executed:

| | |
|---|---|
| 0 | Successful **DISCON** |
| non-0 | Failed **DISCON** |

## Example

```
.
.
CONNECT (H1)            /* connect to synchronous host session 1
SHOW (H1)               /* display synchronous host session 1
.
.
CONNECT (H2)            /* connect to synchronous host session 2
SHOW (H2)               /* display synchronous host session 2
.
.
CONNECT (A1)            /* connect to asynchronous host session 1
SHOW (A1)               /* display asynchronous host session 1
.
.
CONNECT (H3)            /* connect to synchronous host session 3
SHOW (H3)               /* display synchronous host session 3
.
.
DISCON (H2)             /* synchronous host session 2 terminated,
                        /* synchronous host session 3 remains
                        /* connected session
.
.
DISCON (H3)             /* synchronous host session 3 terminated,
                        /* synchronous host session 1 automatically
                        /* connected
.
.
DISCON (H1)             /* synchronous host session 1 terminated,
                        /* asynchronous host session 1 automatically
                        /* connected
.
.
DISCON (A1)             /* asynchronous host session 1 terminated,
                        /* local session 1 automatically connected
                        /* and displayed
.
.
```

# DUP

**Purpose** Simulates the action of the duplication key on the keyboard.

**Format** [label:] DUP

**Remarks** This command is effective in synchronous sessions.

## Example

```
CURSOR (5,11)         /* position cursor to first field
TEXT ("Order 125")    /* enter some text
TAB(2)                /* tab to third field on screen
DUP                   /* duplicate text
```

# EJECT

**Purpose** Inserts a form feed character in the print file that causes a page eject when the file is printed.

**Format** [label:] EJECT

**Remarks** The **EJECT** command applies to the MS-DOS version of ESCORT and is not available in the UNIX operating system version of ESCORT. It is included for script compatibility between the UNIX operating system version and MS-DOS operating system version of ESCORT.

**Example** This example uses the **EJECT** command to print each of five records on a new page.

```
CHAR (80)  REC(5)
     .
     .   (more code)
     .
FOR J = 1 TO 5
DO
    EJECT              /* execute form feed on printer
    PRINT REC (J)      /* print record on new page
ENDO
```

# ENDP

**Purpose** Indicates the end of a program.

**Format** ENDP

**Remarks** Only one **ENDP** (end of program) statement is allowed in a program. It is complementary to the **PROG** statement, which indicates the beginning of a program. Upon execution of the **ENDP** statement in a program, a zero exit code is returned to the UNIX shell.

**See also** **PROG** statement.

**Example**

```
pl  PROG  sl        /* beginning of program 1
     .              /* global section
     .
sl  SCRIPT          /* beginning of script 1
     .
     .
    ENDS            /* end of script sl
    ENDP            /* end of program pl
```

# ENDS

**Purpose** Indicates the end of a script section in a program.

**Format** [label:] ENDS

**Remarks** There must be an **ENDS** statement for each corresponding **SCRIPT** statement. The **ENDS** statement also functions as a **RETURN** command by returning control back to the calling script (or to the shell that initiated ESCORT if **ENDS** is encountered in the main script).

**See also** **RETURN** command and **SCRIPT** statement.

**Example**

```
p1  PROG  s1       /* beginning of program 1
    .              /* global section
    .
s1  SCRIPT         /* beginning of script 1
    .
    .
    ENDS           /* end of script s1
    ENDP           /* end of program p1
```

# ENTER

**Purpose** Simulates the action of the enter key on the keyboard.

**Format** [label:] ENTER

**Remarks** This command is effective in synchronous and asynchronous sessions.

After an **ENTER** command is executed, script execution is suspended until the keyboard is unlocked.

**See also** AID command.

## Example

```
TEXT "Place text here"
TAB (2)                         /* tab over two fields
TEXT "Place additional text here"
ENTER                           /* send information on screen to host
```

# ERASEW

**Purpose** Removes all existing windows.

**Format** [label:] ERASEW

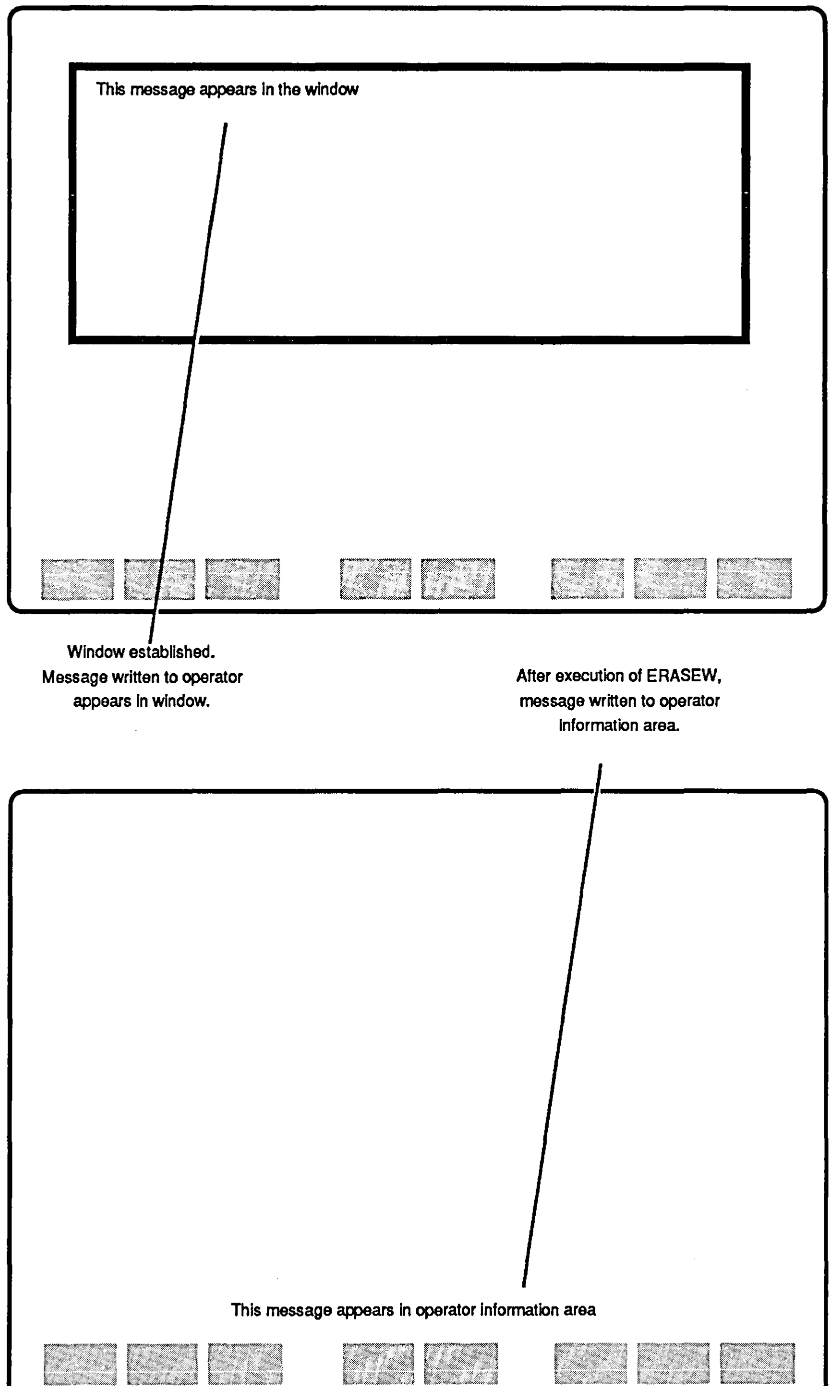
**Remarks** The **ERASEW** command is the only way to remove a resident window.

Any subsequent **WTO** (Write To Operator) message is written to the default **WTO** area, the operator information area, until you establish a new window.

**See also** **WINDOW** and **WTO** commands.

**Example** The following example shows you how to use the **ERASEW** command to remove a resident window.

```
WINDOW (3,5,15,75,R)        /* establish resident window
WTO "This message appears in the window."
ERASEW                      /* remove window
WTO "This message appears in operator information area."
```


This message appears in the window
Window established.
Message written to operator
appears in window.
After execution of ERASEW,
message written to operator
information area.
This message appears in operator information area

# ERIN

**Purpose** Simulates the action of the erase-input key on the keyboard.

**Format** [label:] ERIN

**Remarks** This command is effective in synchronous and local sessions.

This command clears all the unprotected data fields on your current screen.

## Example

```
PF4          /* add a record
  .
  .
  .
ERIN         /* erase all unprotected data on screen
```

# EROF

**Purpose** Simulates the action of the erase-to-end-of-field key on the keyboard.

**Format** [label:] EROF

**Remarks** This command is effective in synchronous and local sessions.

This command removes all unprotected data beginning at the current cursor position until the end of the field.

**Example**

```
 .
 .
PF4         /* add a record
TAB(2)      /* tab to third field on screen
EROF        /* erase the third field
 .
 .
```

# EXIT

**Purpose** Exits from script mode to either Interactive mode or Tutorial mode.

**Format** [label:] EXIT [(TUTORIAL)]

TUTORIAL specifies exit to Tutorial mode from script mode when connected to an active host session. If you do not specify the keyword *TUTORIAL*, ESCORT will default to:

- Interactive mode from script mode when connected to an active host session.
- Tutorial mode from script mode when connected to a local session.

**Remarks** The **EXIT** command can be used for two main purposes: exiting from a script and entering Interactive mode to allow for data entry, or entering Tutorial mode.

**Using EXIT to Enter Interactive Mode**

The **EXIT** command is useful during script execution, when connected to an active host session, since it enables you to exit the script and enter Interactive mode. Once in Interactive mode, data can be entered. Script execution is resumed by using the Interrupt/Resume (I/R) key sequence, ( ESC **f 2**).

The ability to enter data in Interactive mode can be used, for example, to attempt to recover from an error condition.

Another common use of the **EXIT** command is to help you debug scripts. You can use the **EXIT** command to insert break points in a script. Once you complete the debugging stage, you may remove the **EXIT** commands from the script.

When an **EXIT** command is encountered in a script (or when script execution is interrupted manually by pressing ESC **f 2**), script execution

is suspended and Interactive mode is entered. You may perform as many transactions as you wish while in Interactive mode. In order to resume script execution, press the I/R key sequence, ESC **f 2**.

**Using EXIT to Enter Tutorial Mode.**

The other main use of the **EXIT** command is to enter Tutorial mode. You may use the keyword *TUTORIAL* in the **EXIT** statement when connected to an active host session, or the **EXIT** statement without the keyword when connected to a local session, to enter Tutorial mode. Tutorial mode enables you to use ESCORT as an on-line tutorial for your application. Tutorial mode suspends script execution temporarily so that data can be entered from the terminal. It differs from Interactive mode because pressing any AID key will resume script execution.

You can use this feature most effectively in combination with the **WINDOW** and **WTO** commands. These commands enable you to send instructions or messages to an operator before entering Tutorial mode. For example, you can have the script prompt an operator to enter information such as a password or request input parameters to be used later in the script.

When you enter Tutorial mode, script execution is suspended until an AID key is pressed. However, neither the data nor the AID key is sent to the host system at that point. When the AID key is pressed, script execution resumes, but it is up to the script to decide whether or not to send the entered data and AID key to the host. Therefore, you have the ability to perform edit checks on the entered data and AID key before they are sent to the host system. A system global integer variable, *SYSAID*, is provided to check which AID key has been entered. The table below lists the AID keys and the corresponding *SYSAID* values.

| AID key | SYSAID value |
|---|---|
| ENTER | 0 |
| PF1 | 1 |
| PF2 | 2 |
| . | . |
| . | . |
| PF24 | 24 |
| CLEAR | 25 |
| PA1 | 26 |
| PA2 | 27 |
| PA3 | 28 |
| ATTN | 29 |
| SYS_REQ | 30 |

The only AID keys available in the asynchronous environment are:

ENTER
PF1 - PF8 (corresponding to soft function keys F1 to F8 )
CLEAR

All other AID keys are ignored.

To avoid leaving your terminal in either Interactive or Tutorial modes for an indefinite period, a time-out value can be specified using the *EXIT* keyword in a **TIMEOUT** command. When the time-out value expires control is returned to the script.

**See also** **TIMEOUT, WINDOW** and **WTO** commands.

**Example 1** This example prompts a user through a login procedure. The **EXIT** command enables the operator to enter a user ID and password in Interactive mode. Program execution continues after ESC **f 2** is pressed. If the login fails, the **EXIT** command is used again to enter Interactive mode for another try.

```
TEXT        "/for login"
ENTER                       /* get login screen
WTO    "ENTER USERID AND PASSWORD THEN PRESS ESC f 2"
EXIT                        /* enter interactive mode
       /***continue here after ESC f 2 is pressed***/
WHILE  $SCAN ("LOGIN FAILED"(24,1,80))
```

```
DO
    WTO   "TRY AGAIN"
    EXIT                    /* enter interactive mode
ENDO
```

**Example 2** This example calls an error routine if the operator does not enter the date and press PF1.

```
WTO   "ENTER DATE THEN HIT PF1"
EXIT  (TUTORIAL)                     /* enter tutorial mode
         /***continue here after AID key is pressed***/
IF    (SYSAID != 1) |                /* PF 1?
      (DATE != CURDATE) THEN         /* correct date?
      CALL ERROR
ENDIF
```

# FIELD

**Purpose**

The first format assigns a symbolic name (a screen field variable) to a specified area on the screen.

The second format assigns a symbolic name to and defines the attributes for a specified area within a local screen format.

**Format**

**FIELD (row,col,length) [format.]field_name**

*or*

```
FIELD   (row,col,length, (attr_list))
        {[format.]field_name}  ["char_string"]
        {DUMMY              }
```

**row, col** specifies the screen address of the field. The *row* can be between 1 and 24; and the *col* can be between 1 and 80, inclusive. The *row* and *col* must be integer constants.

**length** specifies a length for the field. The *length* must be between 1 and 1919, inclusive (the entire screen). The *length* must be an integer constant.

**attr_list** specifies the attributes for the field. There are seven groups of attributes. Groups 1 to 4 are the Primary Attributes, groups 5 and 6 are the Extended Field Attributes and group 7 is an additional attribute, not provided by IBM, that specifies background color.

You must select one attribute from each group or select the default attribute by using an asterisk (*). Each attribute must be separated by a comma, a blank space or a tab.

The following tables list the seven groups of Primary Attributes and Extended Field Attributes.

| **Primary Attribute - Group 1** | |
|---|---|
| Attribute | Code |
| Protected | P |
| Unprotected | U |
| Default (U) | * |

| **Primary Attribute - Group 2** | |
|---|---|
| Attribute | Code |
| Numeric | N |
| Alphabetic | A |
| Default (A) | * |

| **Primary Attribute - Group 3** | |
|---|---|
| Attribute | Code |
| Normal | N |
| Highlighted | H |
| Dark | D |
| Default (N) | * |

| **Primary Attribute - Group 4** | |
|---|---|
| Attribute | Code |
| Modified DT | M |
| Reset DT | R |
| Default (R) | * |

| **Extended Field Attribute - Group 5** | |
|---|---|
| Attribute | Code |
| Normal | N |
| Blink | B |
| Reverse video | R |
| Underline | U |
| Default (N) | * |

| Extended Field Attribute Foreground - Group 6 | |
|---|---|
| Attribute | Code |
| Black | 0 |
| Blue | 1 |
| Green | 2 |
| Cyan | 3 |
| Red | 4 |
| Magenta | 5 |
| Brown | 6 |
| White | 7 |
| Gray | 8 |
| Light blue | 9 |
| Light green | 10 |
| Light cyan | 11 |
| Light red | 12 |
| Light magenta | 13 |
| Yellow | 14 |
| Hi-lit white | 15 |
| Default (7) | * |

| Extended Field Attribute Background - Group 7 | |
|---|---|
| Attribute | Code |
| Black | 0 |
| Blue | 1 |
| Green | 2 |
| Cyan | 3 |
| Red | 4 |
| Magenta | 5 |
| Brown | 6 |
| White | 7 |
| Default (0) | * |

**format.** specifies an optional screen format name to identify uniquely a field name that may appear in multiple formats.

**field_name** specifies the simple field name for a particular format. For more information on naming conventions for field names see the section, "Field Variables", in Chapter 2.

**DUMMY** is a keyword that allows you to declare a literal field, or a field which you are not going to access by a symbolic field variable name.

**char_str** pre-initializes a field. A Protected field can be initialized with a literal character string. An Unprotected data entry field can be initialized with an integer constant or a string constant, depending upon the attribute selected from the options in Primary Attribute - Group 2 . If the *char_str* operand is omitted the field is initialized with blanks.

**Remarks**

Assigning data to a field is the same as setting the cursor to the field position and using the **TEXT** command. For example, the following two statements are equivalent:

Statement 1

```
FIELD (5,10,8)  ATT.USERID
   .
   .
ATT.USERID = "ORDERXYZ"
```

Statement 2

```
CURSOR  (5,10)
TEXT    "ORDERXYZ"
```

Both of these statements position the cursor on the screen at row 5, column 10 and enter a 8 character string, "ORDERXYZ" on the screen.

Using field names can help you maintain your scripts. If a script contains **CURSOR** commands, it is necessary to update the row and column values in the script if the screen's field definitions change. You would also have to remember which scripts are affected by a particular changed screen. By using symbolic names you can avoid this time consuming work.

It is good programming practice to define fields in an external file. The external file is included in the required scripts by using the **COPY** command. Changes to **FIELD** statements, used in multiple scripts, need only be made once.

A screen field variable may be used wherever a string variable is appropriate in a program.

Any null values within a field are converted to blanks.

Every local screen field created with a **FIELD** statement contains an attribute byte. Each attribute byte occupies one character position on the screen, located at the first position in each field. It is important to take this into account when defining the starting column position for a field.

In the following example, field *A* is defined with a length of 10 characters and starts in row 5 at column 2; the position in row 5 at column 1 will be occupied by the attribute byte for field *A*. Similarly, field *B*, which starts immediately after field *A* begins at column 13 since the position in row 5 at column 12 will be occupied by the attribute byte for field *B*.

```
.
.
FIELD (5,2,10) A
FIELD (5,13,7) B
.
.
```

Refer to Appendix D for information on interpreting the attribute byte.

A script, *fldgen*, is provided on your ESCORT installation diskette to assist you in creating field variables for a given synchronous host screen. For more information on this script, see the section, "Generating Screen Field Variables", in Chapter 5. Refer also to the "Local Screen Generator Utility Program" described in Chapter 6, for information on creating local screens.

Of the two **FIELD** statement types, the first type is used primarily to declare field variables for formatted screens in a host session. Within this **FIELD** statement type, there are two kinds of screen field variables: *specific* and *common.*

### Specific Screen Field Variables

A specific screen field variable declares a field on a specific screen. The format of the specific screen field is:

```
FIELD  (row, col, length)  format.field
```

Examples of specific screen field variable declarations follow:

```
FIELD  (7,25,7)  ORDER.USERID
FIELD  (4,13,12) CUST.CUSTID
```

### Common Screen Field Variables

A common screen field variable declares a field common to more than one screen. The format of the common screen field is:

```
FIELD  (row, col, length)   cfield
```

The cfield (common field) specifies the common screen field name. The common field name applies to any screen format in the program (it has the same location, length, and name on every screen). A common field name may have from 1 to 8 characters and the first character must be alphabetic.

Examples of common screen field variable

declarations follow:

```
FIELD (24,1,80)  sys_error
FIELD (24,60,20) actionmsg
```

The second **FIELD** statement type is used to create formatted screens for local sessions. Use of the Primary Attributes and Extended Field Attributes allow you to create screens that contain almost all of the attributes of an actual host application screen.

**Note**
The foreground and background colors, defined by the Extended Field Attribute, groups 6 and 7, are not available in the UNIX operating system version of ESCORT. They are defined for script compatibility between the UNIX operating system version and MS-DOS operating system version of ESCORT.

**FIELD** statements of the second type must be defined in a local screen format definition area that starts with a **BEGFMT** statement and ends with an **ENDFMT** statement. For more information on defining this second type of **FIELD** statement see the section, "Local Session Screens", in Chapter 2.

ESCORT does not support light-pen-detect among the Primary Attributes, nor Base Character Set among the Extended Field Attributes, and none of the Extended Character Attributes.

See also **BEGFMT/ENDFMT**, **FORMAT** and **TEXT** commands.

**Example 1** This example demonstrates the declaration of global and local field variables using the first type of **FIELD** statement.

```
p1    PROG       S1
      /***  Global variable declarations   ***/
      .
      .
      /***  Screen field global declarations ***/
      FIELD (24,1,80)  J          /* field J
      FIELD (1,60,20)  K          /* field K
      FIELD (1,60,20)  F.A        /* format F, field A
      FIELD (24,1,80)  G.K        /* format G, field K
S1    SCRIPT
      /***  Local variable declarations   ***/
      .
      .
      /***  Screen field declarations     ***/
      FIELD (5,8,11)   A.A        /* format A, field A
      FIELD (9,9,12)   A.B        /* format A, field B
      FIELD (5,8,19)   C.A        /* format C, field A
      FIELD (5,20,10)  C.B        /* format C, field B
      .
      FIELD (9,10,16)  X.Y        /* format X, field Y
      .
      .
```

**Example 2** In this example, two literal fields named *USER ID:* and *PASSWORD:*, and two screen field variables named *user_id* and *passwd* are declared for a local screen format called *logon*, using the second type of **FIELD** statement.

```
p2     PROG       s1
       /*** Global variable declarations ***/
       .
       .
       /*** Local screen formats ***/
       BEGFMT logon
         FIELD (5,12,8,(P,*,H,*,*,*,*)) DUMMY "USER ID:"
         FIELD (5,21,6,(*,*,*,*,*,*,*)) usr_id
         FIELD (10,12,9,(P,*,H,*,*,*,*)) DUMMY "PASSWORD:"
         FIELD (10,22,8,(*,*,D,*,*,*,*)) passwd
       ENDFMT
s1     SCRIPT
       .
       .
```

# FM

**Purpose**

Simulates the action of the field-mark key on the keyboard.

**Format**

[label:] FM

**Remarks**

This command is effective in synchronous sessions.

If you are using TSO/SPF, you may enter multiple commands on a single line if you use the field-mark key.

**Example**

This example demonstrates the use of the **FM** command to execute a logoff procedure in TSO/SPF.

```
      .          /* you are on EDIT screen
      .
TEXT  "=x"        /* get out of SPF
FM                /* field mark
TEXT  "logoff"    /* logoff command
ENTER             /* execute both commands
```

## FOR

**Purpose** Enables execution of a block of commands in a *DO/ENDO* loop for a specified number of times.

**Format**

```
[label:]  FOR var=init TO final [STEP incr]
          DO
            .
            .
            .
           statement(s)
            .
            .
            .
          ENDO
```

**var** specifies a counter. The *var* must be an integer variable.

**init** specifies the initial value of the counter. This can be an integer or integer expression.

**final** specifies the final value of the counter. This can be an integer variable or an integer constant and may be either positive or negative. The maximum absolute value of *final* is $2^{31} - 1$. Execution of the program continues after the **ENDO** statement once the value of *final* has been passed.

**incr** specifies a value by which to increment the counter (or decrement it if the value is negative). The value is added to the *init* field after each pass through the *DO/ENDO* loop. The *incr* value can be either an integer constant or an integer variable. The default value is +1.

**Remarks**

Following is a listing of the execution process of the loop:

1. The counter (*var*) is set to the initial value you specified (*init*).
2. The current value of the counter is compared with the *final* value. If the counter has an absolute value greater than the absolute value of the *final* value, the program branches out of the *DO/ENDO* loop.
3. The statements following the **DO** statement are executed sequentially until the **ENDO** statement is reached.
4. The counter is incremented (or decremented) by the value you specified in *incr*.
5. Control is transferred to Step 2.

These are some general rules for loops:

- You may use nested **FOR** statements in your program.
- The **DO** and **ENDO** statements are required in your program even if you execute only one statement or no statement at all.
- You may use the **BREAK** and **CYCLE** commands within your *DO/ENDO* loop to change the path of execution of the loop.

**See also**

**BREAK** and **CYCLE** commands.

**Example 1**

This program section shows how to use the **FOR** statement to print an array containing 5 elements.

```
CHAR (80)  REC(5)
    .
    .   (more code)
    .
FOR J = 1 TO 5
DO
   PRINT REC (J)
ENDO
```

**Example 2** In this example, 25 elements of a string array, *names*, are assigned from another array, *source*. If the name in *source* is *Hangman*, there is no assignment. Each element of the array is a 20 character string. Nested **FOR** loops are shown.

```
CHAR (20)  names (25)
CHAR (20)  source (5)
INT        i
INT        j
INT        k

source = ("Dude", "Badguy", "Hangman", "Henchman", "Toad")

FOR i=1 TO 5                  /* automatically defaults to increment of 1
DO
   FOR j=1 TO 5    STEP 1     /* counter explicitly incremented by 1
   DO
      IF source (j) = "Hangman"
      THEN CYCLE               /* goes to first ENDO
      ENDIF
      k = ((i-1)*5 + j)       /* next element subscript
      names (k) = source (j)
   ENDO
ENDO
```

# FORMAT

**Purpose** Defines the current default format name.

**Format** [label:] **FORMAT format**

**format** specifies the new default format name.

**Remarks** The *format* operand in a **FORMAT** statement is defined in a **FIELD** statement. The **FORMAT** command is a preprocessing command that relieves you of the laborious task of coding format-qualified field names (specific screen field names). Field names preceded by a dot ( . ) in the statements following the **FORMAT** command are automatically prefixed with the specified format name.

**See also** **FIELD** statement.

**Example 1** The first program does not use the **FORMAT** command. This program requires entry of full screen-field names in assignment statements.

```
TEXT  "/FOR ORDER"                  /* get MFS format
ENTER
ORDER.ORDERNO = "00000034"          /* explicit qualification
ORDER.CLEXTID = "USNENJFJ"
ORDER.INSTID  = "USNENJFJ"
ORDER.REQDATE = "052986"
PF4
IF    !($SCAN ("ADD COMPLETED" (24,1,80)))
THEN  WTO  "FAILED TO ADD ORDER"
      EXIT
ENDIF
```

**Example 2** This example is similar to Example 1, except that the **FORMAT** command is used. Note the use of the dot in the field names.

```
TEXT  "/FOR ORDER"                 /* get MFS format
ENTER
FORMAT ORDER                       /* set default qualifier
  .ORDERNO = "00000034"             /* implicit qualification
  .CLEXTID = "USNENJFJ"
  .INSTID  = "USNENJFJ"
  .REQDATE = "052986"
PF 4
IF    !($SCAN ("ADD COMPLETED" (24,1,80)))
THEN  WTO "FAILED TO ADD ORDER"
EXIT
ENDIF
```

**Example 3** This example illustrates the use of the **FORMAT** command to code the fields for customer and supplier screens.

```
FORMAT  customer
  .name   = "Brown's Shoes"
  .phone  = "111-3333"
  .acct   = "123400"
  .street = "Roosevelt Drive"
  .city   = "Pasadena"
  .state  = "CA"
FORMAT  supplier
  .name   = "Footwear Manufacturing Co."
  .phone  = "222-3333"
  .acct   = "5432100"
  .street = "Elm Street"
  .city   = "Springfield"
  .state  = "Il"
```

# FRESH

**Purpose** Updates the ESCORT screen buffer with data from the host.

**Format** [label:] FRESH

**Remarks** This command is effective in synchronous and asynchronous sessions.

The **FRESH** command is especially useful in the following types of synchronous host applications:

- A screen appears in pieces.
- You expect to receive multiple messages (e.g., TSO logon).
- You expect the host response after an unpredictable number of intermediate responses (e.g., IMS no-response mode transactions).

In the asynchronous environment data is displayed at the terminal following a **WAIT** or a **FRESH** command only. The **FRESH** command provides up to 24 screen-lines of asynchronous host data.

**See also** **$SCAN** function and "AID Subroutines Library" in Appendix C.

**Example** This example demonstrates use of the **FRESH** command during login to IMS or TSO when unpredictable or multiple synchronous host responses are received.

```
      .
      .
TEXT  "IMS"
ENTER
/******** loop until expected message arrives
WHILE !($SCAN ("IMS/VS SIGNON SCREEN"))
DO
      FRESH       /*read SCREEN buffer and display
ENDO
      .
      .
```

```
TEXT  "TSOOM USERID"
ENTER
/******** loop until expected message arrives
WHILE !($SCAN ("ENTER AN 'S' BEFORE EACH OPTION DESIRED"))
DO
      FRESH
ENDO
      .
      .
```

# GETFMT

**Purpose** Loads a specified local screen format into a given local session's presentation space.

**Format** [label:] **GETFMT (local_session-id, screen_name)**

**local_session-id** specifies the local session identification. Valid *local_session-ids* are:

| | |
|---|---|
| L1 | Local session 1 |
| L2 | Local session 2 |

**screen_name** specifies the local screen format name. The local screen format name is defined by the **BEGFMT** statement in the local screen format definition area.

**Remarks** This command is effective in local sessions.

Only one local screen format can be loaded into the local session's presentation space at any given time.

**See also** **BEGFMT/ENDFMT** statement.

**Example** In this example, local session L1 is used to deal with an order entry system using the *order* format, and local session L2 is used to deal with an inventory control system using the *stocking* format.

```
.
.
GETFMT (L1, order)                      /* load order format in L1
GETFMT (L2, stocking)                   /* load stocking format in L2
CONNECT (L1)                            /* connect to local session
SHOW (L1)                               /* display local session
EXIT                                    /* go to interactive for data
.
.
CONNECT (L2)                            /* connect to local session
SHOW (L2)                               /* display local session
EXIT                                    /* go to interactive for data
.
.
```

# GOTO

**Purpose**
Changes the script execution path by unconditionally branching to another statement within the script.

**Format**
**[label:] GOTO label**

**label** specifies a label within the current script section.

**Remarks**
It is good programming practice to use the **BREAK** command rather than the **GOTO** command to branch out from a *DO/ENDO* loop.

**See also**
**BREAK** and **CYCLE** commands.

**Example**

```
test  PROG   test
      .
test  SCRIPT
      .
logl: TEXT  "/FOR LOGON"
      ENTER
      .
      GOTO end              /* branch forward
      .
      GOTO logl             /* branch backward
end:  WTO   "TEST END = 0"
      .
      .
      ENDS
      .
      .
      ENDP
```

## HOME

**Purpose** Simulates the action of the home key on the keyboard.

**Format** [label:] HOME

**Remarks** Use of the **HOME** statement positions the cursor at the first unprotected field on the screen.

**Example** This example prints a PDS member named *MFS* using the ISPF 3.6 option on TSO.

```
  .
  .
TEXT "ISPF"
ENTER
TEXT "3.6"
ENTER
TEXT "JCL.CNTL(MFS)"
HOME                   /* positions the cursor at option entry field
TEXT "J"
ENTER
PF3
  .
  .
```

# IF

**Purpose**

Evaluates a relational expression which yields a *true* or *false* condition and allows you to change the script execution path based on the result.

**Format**

```
[label:]   IF   clause
           THEN
             statement(s)
           ELSE
             statement(s)
           ENDIF
```

**clause** specifies an expression that returns a *true* (non-zero) or a *false* (zero) value. The expression may be an integer or relational expression or a combination of these expressions separated by *&* or | operators. A relational expression always returns an integer value (zero for *false*, non-zero for *true*).

**statement(s)** specifies a block of ESCORT code.

**Remarks**

**THEN** and **ENDIF** are required in an **IF** statement. **ELSE** is optional.

You may nest **IF** statements.

Labels are not allowed on **THEN, ELSE**, or **ENDIF**.

When an **IF** statement is encountered in a program, the expression following the **IF** is evaluated. The execution path followed depends upon the value returned from the expression.

If the result of the expression is *true*, the statements following **THEN** are executed until an **ELSE** or **ENDIF** is encountered. If the result of the expression is *false*, the statements following **ELSE** are executed until an **ENDIF** is encountered.

## Example

```
IF  (string1 = "abcdef")      /* string variable/constant
          |                  /* or
    (counter1 = counter2)     /* integer variables
          |                  /* or
    ($Scan ("ADD COMPLETED")) /* field scan
THEN
    .                        /* THEN action statements
ELSE
    .                        /* ELSE action statements
ENDIF
    .
    .
    .
IF  ((a>b) & (c>d))
          |                  /* or
    ((e(5)=2) & (f!="SOS"))
THEN
    .                        /* THEN action statements
ENDIF

/**where:  a and b are integer variables
/**        e is an integer array
/**        c and d are string variables
/**        f is a field variable
```

# INS

**Purpose**

Simulates the action of the insert key on the keyboard and sets the terminal in insert mode.

**Format**

[label:] INS

**Remarks**

This command is effective in synchronous and local sessions.

This command is commonly used to insert data at a particular cursor position. Data to the right of the cursor is shifted right as long as there are nulls at the end of the field.

You can terminate insert mode by using either a **RESET** command or an AID key command (such as ENTER).

**See also**

**RESET** and **DEL** commands.

**Example**

This example demonstrates use of the **INS** command along with the **TEXT** command to insert data.

```
INS                   /* puts terminal in insert mode
CURSOR (12,5)         /* puts cursor at row 12, column 5
TEXT "inserted text"  /* text inserted at row 12, column 5
RESET                 /* ends insert mode
```

# INT

**Purpose** Declares an integer variable or an integer array.

**Format**

INT **name**
*or*
INT **name (#elements)**

**name** specifies the name of the variable. The variable name must not be a reserved word.

**#elements** specifies the number of elements in an array. You may specify any number of elements from 1 to 2048, inclusive.

**Remarks** The **INT** statement allocates a storage area in memory and assigns a symbolic name to the storage area for an integer or integer array.

The integer (or integer element) may be assigned any value between $-2^{31}+1$ and $+2^{31}-1$, inclusive.

When an integer is declared, it contains a zero value.

**See also** **ASSIGN(=)**, **CHAR**, and **FIELD** statements.

## Example

```
INT table(100)     /* declares an integer array of 100 elements
INT i              /* declares an integer variable "i"
INT j              /* declares an integer variable "j"
INT k              /* declares an integer variable "k"
        .
        .
j = -1
        .
        .
k = (j - 5)
        .
        .
FOR  i=1 TO 100  /* initialize integer array
DO
     table(i) = i
ENDO
        .
        .
```

# LBREAK

**Purpose** Simulates the action of the line-break key on the keyboard.

**Format** [label:] LBREAK

**Remarks** This command is effective in asynchronous sessions.

**Example**

```
    .
    .
CONNECT (A1)
ENTER
WAIT (1)                          /* wait for one second
LBREAK
WAIT (1, "SIGNON", "DISCONNECT")  /* wait for prompts for one second
    .
    .
```

## LOG

**Purpose** Specifies data to be written to the log file.

**Format** [label:] LOG {str_expr}
{SCREEN}

**str_expr** specifies data to be written to the log file. The *str_expr* may contain a string expression that includes a string constant, string variable, string array element, screen field variable, or string function. It may also be a combination of any of the above types of operands separated by a concatenation operator. If you use more than one constant or variable, you must enclose the expression in parentheses.

**SCREEN** is a keyword used to write the current screen image (1920 characters) to the log file. It is a system global variable.

**Remarks** The log file contains data or messages defined by the user.

The ESCORT log file name is *escort.lg{proc-id}*, where *{proc-id}* refers to the unique process identification the UNIX operating system assigns to each process. The file is created in the directory defined by the ESCDIR environment variable.

Checking for a successful **LOG** operation, when the *escort.lg{proc-id}* file is first created, is good programming practice. The internal global integer variable, *SYSRET*, returns the result of a **LOG** operation. *SYSRET* may have one of the following values after the **LOG** is executed:

| | |
|---|---|
| 0 | Successful **LOG** |
| non-0 | Failed **LOG** |

The command will fail if the output file, *escort.lg{proc-id}* cannot be created. A message

will be written to the *escort.pr{proc-id}* file.

**See also** CHKPT, PRINT, and WTO commands.

## Example

```
    .
    .
LOG  LINE24      /* global field variable
LOG  "case 20"   /* simple string constant
LOG  ($TIME + " TEST002 COMPLETED SUCCESSFULLY")
LOG  SCREEN      /* log current screen image
LOG  ("Code " $ITOS(i) "=" codetype (i))
```

# NL

**Purpose** Simulates the action of the new-line key on the keyboard.

**Format** [label:] NL [(n)]

**n** specifies the number of lines you want to skip over. The *n* can be either an integer constant or an integer variable with a value between 1 and 64 inclusive. The default value for *n* is 1.

## Example

```
CURSOR (12,34)   /* positions cursor at row 12, col 34
TEXT   "some data"
NL     (3)       /* skips 3 lines
TAB    (4)       /* tabs over 4 fields
TEXT   "more data"
```

# OPEN

**Purpose** Opens a file in order to read, write, or append data.

**Format** [label:] OPEN (nickname, filename, {R})
{W}
{A}

**nickname** specifies the internal (ESCORT) name of the file. It must be declared in the **OPEN** command. The *nickname* must be 1 to 8 characters, and the first character must be alphabetic. This name is used in the **READ**, **WRITE**, **CLOSE**, and **CHKPT** commands. The *nickname* is global and can be used in any script within the entire program.

In an ESCORT program, you may specify up to 10 files, which may be open at the same time.

**filename** specifies the name of the disk file. The *filename* may be a string constant or a string variable. Up to 100 characters are allowed in a *filename*. It may contain the full path name.

{R}{W}{A} specifies the read, write, or append attribute. Append mode permits addition of new data to the end of an existing file. If you specify a {*W*} or {*A*} attribute and the file does not exist, it is automatically created.

**Remarks** Checking for a successful **OPEN** operation is good programming practice. The internal global integer variable, *SYSRET*, returns the result of an open operation. *SYSRET* may have one of the following values after the **OPEN** is executed:

| | |
|---|---|
| 0 | Successful **OPEN** |
| −1 | Failed **OPEN** |

If a file is opened as a pipe between scripts, the file must first have been created as a named pipe

using the UNIX *mknod* system call.

Scripts are portable between the UNIX operating system version and the MS-DOS operating system version of ESCORT and you may, therefore, substitute the standard UNIX operating system slash character (/) in a UNIX file pathname with the MS-DOS back-slash (\) file name separation character when using the **OPEN** command.

See also — **CHKPT**, **CLOSE**, **READ**, and **WRITE** commands.

**Example** — This example opens a file called */usr/myname/myfile* and assigns the nickname *F*. A status check using *SYSRET* is made after the file is opened. If the **OPEN** failed (SYSRET = −1), a message is displayed on the terminal and the program exits to Interactive mode.

```
OPEN    (F, "/usr/myname/myfile", R)      /* open file for READ
IF      SYSRET < 0                      /* check value of SYSRET
THEN
        WTO  "FAILED TO OPEN FILE /usr/myname/myfile"
        EXIT                            /* exit to interactive mode
ENDIF
```

# PAn

**Purpose**

Simulates the action of one of the Program Attention keys (PA1, PA2 or PA3) on the keyboard.

**Format**

[label:] **PAn**

**n** specifies the number of the PA key being simulated. The *n* may have a value of either 1, 2 or 3 representing the PA1, PA2 or PA3 key.

**Remarks**

This command is effective in synchronous sessions.

After a **PAn** command is executed, script execution is suspended until the keyboard is unlocked.

**See also**

**AID** command.

**Example**

```
   .
   .
PA2            /* clear IMS message queue
PA2            /* before getting new format
TEXT  "/FOR FORMATX"
ENTER
   .
   .
```

# PFn

**Purpose**

Simulates the action of one of the 24 Program-Function keys on the 3278 keyboard, or the action of one of the eight soft function keys on the VT100 keyboard.

**Format**

[label:] **PFn**

**n** specifies the number of the PF key being simulated. In a synchronous session, the *n* may have a value of 1 to 24, representing keys PF1 through PF24 on the keyboard.

In an asynchronous session, the *n* may have a value of 1 to 8, representing keys PF1 through PF8 (corresponding to soft function keys F1 to F8 ) on the keyboard.

**Remarks**

This command is effective in synchronous and asynchronous sessions.

After a **PFn** command is executed, script execution is suspended until the keyboard is unlocked.

**See also**

AID command.

**Example**

```
  .
  .
PF4                /* add a record to your file
TAB(2)             /* tab to third field on screen
TEXT "Enter data"
ENTER              /* send data to host
  .
  .
```

# PRINT

**Purpose** Sends data to the ESCORT print file.

**Format** [label:] PRINT {str__expr}
{SCREEN}

**str__expr** specifies data to be written to the print file. The *str__expr* may contain a string expression that includes a string constant, string variable, string array element, screen field variable, or string function. It may also be a combination of any of the above operands separated by a concatenation operator. If you use more than one constant or variable, you must enclose the expression in parentheses.

**SCREEN** is a keyword that prints the current screen image (1920 characters). It is a system global variable.

**Remarks** The ESCORT print file is named *escort.pr{proc-id}* where *{proc-id}* refers to the unique process identification that the UNIX operating system assigns to each process. The print file is created in the directory defined by the ESCDIR environment variable.

**Example**

```
PRINT    LINE24    /* global field variable
PRINT    "Hello"   /* print string constant
PRINT    ($TIME + " TEST002 COMPLETED SUCCESSFULLY")
PRINT    SCREEN    /* print current screen image
PRINT    ("Code " $ITOS(i) "=" codetype (i))
```

# PROG

**Purpose**

Indicates the beginning of a program, defines the name of the program and first script to be executed, and specifies which session is to be connected.

**Format**

**prog_name PROG script_name [(session-id)]**

**prog_name** specifies the name of the program.

**script_name** specifies the name of the first script to be executed when the program begins. Normally, this is the main script.

**session-id** specifies the session identification that will be connected and active when ESCORT is started. Valid *session-ids* are:

| | |
|---|---|
| H1 | Synchronous host session 1 |
| H2 | Synchronous host session 2 |
| H3 | Synchronous host session 3 |
| H4 | Synchronous host session 4 |
| A1 | Asynchronous host session 1 |
| A2 | Asynchronous host session 2 |
| A3 | Asynchronous host session 3 |
| A4 | Asynchronous host session 4 |
| L1 | Local session 1 |
| L2 | Local session 2 |

The *session-id* is an optional operand; if it is omitted ESCORT will connect the default, synchronous host session, H1.

If the *session-id* is specified as an asynchronous session, ESCORT connects to the screen buffer of the specified session; a **CONNECT** command, preceded by a **SERINIT** statement, is required to physically connect to the asynchronous session.

**Remarks**

The **PROG** statement marks the beginning of both the global variable declaration section and the local screen format definition section of the

program. The global declaration and the local format sections are ended by the first script statement. It is good programming practice to define local screen formats immediately after declaring global variables. Except for comments in your program, the **PROG** statement should always be the first statement.

Each program must have at least one script.

A program contains an optional global variable declaration section, an optional local screen format definition section, and one or more scripts. The global variable section may contain only **INT, CHAR,** and **FIELD** statements and comments. The local screen format definition section may contain only **BEGFMT, ENDFMT,** and **FIELD** statements and comments.

You may also use the **COPY** statement in the global variable section to copy code containing declaration statements.

See also **CONNECT, ENDP, SCRIPT** and **SERINIT** commands.

## Example

```
pl    PROG  main (H2)          /* beginning of program statement
                               /* host session 2 connected
       .                       /* global section
       .
      INT i
      CHAR (8)  order
       .
       .
      BEGFMT logon             /* local screen format section
        FIELD (10,12,9,(P,*,H,*,*,*,*)) DUMMY "PASSWORD:"
        FIELD (10,22,8,(*,*,D,*,*,*,*)) passwd
         .
         .
      ENDFMT
       .
       .
main  SCRIPT                   /* beginning of script main
       .
       .
      CALL  s2                 /* call to script s2
       .
       .  (commands and statements)
       .
      ENDS                     /* end of script main
```

```
s2   SCRIPT                  /* start of script s2
      .
      .  (commands and statements)
      .
     ENDS                    /* end of script s2
     ENDP                    /* end of program statement
```

# PROMPT

**Purpose**

Initializes the system global variable, *SYSPRMT*, with an asynchronous host prompt.

**Format**

**[label:] PROMPT (str_expr [,col [,row]])**

**str_expr** specifies the asynchronous host prompt. The *str_expr* may contain a string expression that includes a string constant, string variable, string array element, screen field variable, or string function. It may also be a combination of any of the above types of operands separated by a concatenation operator. If you use more than one constant or variable, you must enclose the expression in parentheses.

**col** specifies the screen column address of the prompt, and can be between 1 and 80 inclusive.

**row** specifies the screen row address of the prompt, and can be between 1 and 24 inclusive.

Both *col* and *row* must be integer constants.

**Remarks**

This command is effective in asynchronous sessions.

The entire operand string from the **PROMPT** command initializes the *SYSPRMT* system global variable. The *SYSPRMT* variable can be used as a parameter to a **WAIT** command which searches for the specified string expression at the defined screen address (if any).

Note that ESCORT provides several address parameter options:

- Column and row screen address are not specified. The **WAIT** command searches the entire data stream for the specified prompt.
- Both column and row screen address are provided. The data stream is searched at the precise screen address for the specified

prompt.

- Only the column screen address is provided. The data stream is searched at the particular column address. This option is useful if, for example, the asynchronous host system response always returns in a specific screen column, as in the case of the UNIX operating system default dollar sign ($) prompt.

Unlike other screen addressing commands, the **PROMPT** command address operands are reversed (i.e., column is defined before row).

See also

**WAIT** command and the section, "Synchronizing Data Transmissions", in Chapter 2.

### Example

```
.
.
CONNECT (A1)            /* connect to asynchronous host A1
PROMPT ("$",1)          /* initialize system prompt in column 1
TEXT "Input to async host"
ENTER                   /* send data to asynchronous host
WAIT (10, SYSPRMT)      /* wait for async host prompt ($),
                        /* time-out and proceed to next command
                        /* after 10 seconds
.
.
```

# PUTENV

**Purpose** Changes or creates a UNIX environment variable.

**Format** **PUTENV (evar=value)**

**evar** specifies the environment variable whose value you wish to change or create.

**value** specifies the value that is assigned to the environment variable.

**Remarks** The *evar=value* operand must be a string constant.

The system global variable, *SYSRET*, returns the result of a **PUTENV** operation. *SYSRET* may have one of the following values after the **PUTENV** operation:

| | |
|---|---|
| 0 | Successful **PUTENV** |
| non-0 | Unsuccessful **PUTENV** |

**See also** **$GETENV** function.

**Example** This example shows how to use **PUTENV** to change the value of the UNIX environment variable *HOME*.

```
CHAR (15) envvar
envvar = "HOME=/usr/xyz"
PUTENV (envvar)
```

# READ

**Purpose** Reads a record from a file.

**Format** [label:] READ (nickname, buffer)

**nickname** specifies an ESCORT internal name for the file. This is the name assigned to the file in the **OPEN** statement. The *nickname* is global and can be used in any script within the entire program.

**buffer** specifies the symbolic name of a string variable to receive the data record. The buffer size should be equal to the maximum record size in the file (maximum possible record size is 2048 characters). Otherwise, data will be truncated and lost.

**Remarks** You must open a file before reading it.

Since tabs are not expanded by ESCORT, be sure you do not inadvertently insert tabs in your file with an editor. ESCORT issues a warning if tabs are encountered, but processing continues.

The input file may have variable length records. The record length may be between 1 and 2048 characters. Records are separated by a new-line character (or a carriage return or both).

The **READ** operation is sequential. When a file is opened, the record pointer points to the first record in the file. **READ** always gets a record from the current pointer and then advances the record pointer to the next sequential record. If a **READ** is attempted after the last record in the file has been read, an end-of-file (EOF) condition is returned.

In order to rewind the file, issue a **CLOSE** command followed by an **OPEN**. This will position the record pointer at the first record in the file again.

Checking for a successful **READ** operation is good programming practice. The internal global variable, *SYSRET*, returns the result of a **READ** operation. *SYSRET* may have one of the following values after the **READ** operation:

| | |
|---|---|
| 0 | Successful **READ**. |
| −1 | Error or end of file encountered. |
| n | Data truncated and lost (*n* is the number of characters returned). |

The length of the data read into the buffer is either the maximum (declared) size of the buffer or the length of the record read. This length may be obtained by using the **$LENGTH** function.

If a file is opened as a pipe between scripts and a **READ** operation is attempted before data has been written to the pipe, an end-of-file condition will be returned in *SYSRET*. In such a case, it may be necessary to include a **WAIT** command to ignore the end-of-file condition. Additionally, a true end-of-file flag must be agreed upon beforehand within the reading and writing scripts. Refer to the "Reading from a Pipe File" script in Appendix G for an example.

See also **CHKPT, CLOSE, OPEN, WAIT,** and **WRITE** commands and the **$LENGTH** function.

**Example** In this example, records are read sequentially from a file (nickname *F*) until the end of file has been reached. The file contains variable length records. The maximum record size in this file is 80 bytes. After a record has been read, the program prints out each record's sequence number, length, and contents. Note that the **$LENGTH** function is used.

```
CHAR (80) buffer        /* 80 byte buffer
INT       length
INT       record

OPEN (F, "FILE1", R)  /* open file
READ (F, buffer)        /* read first record
record = 1              /* set record count = 1

WHILE SYSRET != -1    /* if not EOF then
DO
   length  = $LENGTH(buffer)
   PRINT  ("Record # " + record)
   PRINT  ("Length = " + length)
   PRINT  buffer
      .
      . (more code)
      .
   READ (F, buffer)  /* read subsequent record
ENDO
/*******skips to here when EOF encountered*******/
PRINT "END OF FILE ENCOUNTERED"
```

## RESET

**Purpose** Simulates the action of the reset key on the keyboard.

**Format** [label:] RESET

**Remarks** This command is effective in synchronous sessions.

**See also** INS command.

**Example**

```
INS            /* put terminal in insert mode
TEXT "1238"    /* insert data in field
RESET          /* take terminal out of insert mode
```

# RETURN

**Purpose** Returns control of a program back to the calling script from a subroutine (script) call.

**Format** [label:] RETURN

**Remarks** The **ENDS** statement also functions as a **RETURN** command if the called script does not have a **RETURN** command.

**See also** **ENDS** statement.

**Example** The following example demonstrates a return from a subroutine via a **RETURN** statement and via an **ENDS** statement. The **RETURN** is executed if *a* equals *b* and the **ENDS** is executed if *a* is not equal to *b*.

```
sl      SCRIPT
          .
          .
        IF a=b
        THEN
          RETURN      /* return to calling script if a=b
        ELSE
          .
          .
        ENDIF
          .
          .
        ENDS          /* return to calling script if a!=b
                      /* ENDS functions as a RETURN
```

# RUN

**Purpose** Enables execution of UNIX operating system (or shell) commands from a script.

**Format** **[label:] RUN string**

**string** specifies a string constant or string variable containing the operating system command line. The actual command string length is restricted by the UNIX shell.

**Remarks** You must have a **FRESH** command in your program after the last **RUN** command in order to restore the host screen.

The system global variable, *SYSRET*, returns the result of a **RUN** command. *SYSRET* may have one of the following values after the **RUN** command:

| | |
|---|---|
| 0 | Successful **RUN** |
| −1 | Unsuccessful **RUN** |

Note that a successful **RUN** (*SYSRET* value of 0) does not imply that the command, contained in the *string* operand, executed successfully.

## Example

```
RUN "cls"                 /* clear screen
RUN  "ls /usr/bin"        /* issue list files command
com = "cp filel file2"
RUN  com                  /* issue copy file command
RUN  "userprog"           /* run a user program
FRESH                     /* needed to restore the host screen
```

# SCRIPT

**Purpose** Indicates the beginning of a script section in your program and defines the name of the script.

**Format** **script_name SCRIPT [(decl_list)]**

**script_name** specifies the name of the script. This name may appear in a **PROG** or **CALL** statement. The *script_name* may be up to 8 characters. The first character must be alphabetic.

**decl_list** declares parameters passed on a **CALL** statement. The *decl_list* may contain an integer constant, integer variable, string constant, string variable, screen field variable, or more than one of the above separated by commas (arrays or array elements are not allowed). The *decl_list* is required if parameters are passed to the script on a **CALL** statement, and it must be enclosed in left and right parentheses.

For each entry in the *decl_list* in the **SCRIPT** statement, there must be a corresponding parameter in the *parm_list* in the **CALL** statement. Each type of parameter in the *parm_list* and *decl_list* must be consistent. See the table below for examples:

| **parm_list** | **decl_list** |
|---|---|
| integer constant | integer variable |
| integer variable | integer variable |
| string constant | string variable |
| string variable | string variable |
| field variable | screen field variable or string variable |

**Remarks**

Passed parameters are defined in the **SCRIPT** statement. When a constant is passed, a local variable is allocated and the value of the constant is assigned to it. When a variable is passed, the address and length of the local variable are changed to those of the passed parameter. Therefore, any change to a locally declared variable in the *decl_list* modifies the corresponding variable in the *parm_list*.

If a field variable is passed, the *decl_list* parameter may be a field variable or a string variable. In the first case, the address and length of the passed field variable are assigned to the receiving field variable. In the latter case, a local string is allocated and the contents of the passed field are copied into it.

The row, column, and length of a field variable in the *decl_list* should contain an asterisk (*). Note that only the contents of a field variable are passed, the attributes of the field variable, to which the contents are assigned, are governed by the defined attributes, if any, for that field.

Similarly, the length of a string variable should contain an asterisk (*). For example:

```
S1 SCRIPT (INT i, CHAR (*) buf, FIELD (*,*,*) f1)
```

You may pass a global variable on the call in the *parm_list*. In this case, you must not use the same variable in the *decl_list*. Local variables may have the same name in both the *parm_list* and the *decl_list*.

**See also**

**CALL** and **ENDS** commands.

**Example 1** This example shows the script structure in a program.

```
pl    PROG  main          /* program pl, start script 'main', default session H1
      .
      .                   /* global variable declaration
      .
main  SCRIPT              /* start of script 'main'
      .
      .
      CALL  s2 (parm_list) /* call script s2
      .
      .  (commands and statements)
      .
      ENDS                /* end of script 'main'
s2    SCRIPT (decl_list)  /* start of script s2
      .
      .  (commands and statements)
      .
      ENDS                /* end of script s2
      ENDP                /* end of program pl
```

**Example 2** In this example, four parameters are passed. Two will contain returned values.

```
pgm   PROG    main

main  SCRIPT
      INT        code                  /* output parm - integer
      CHAR (80) response               /* output parm - string
      FIELD (24, 1, 80) (line24)

      CALL       sub (80, code, "ADD COMPLETED", response)
      IF         code = 0
      THEN       PRINT "SUCCESSFUL ADD"
      ELSE       PRINT "ADD FAILED"
      ENDIF
      PRINT      response
      ENDS

sub   SCRIPT     (INT  length,         /* integer constant - input
                  INT  rtncode,        /* integer variable - output
                  CHAR (*) message,    /* string constant - input
                  CHAR (*) response,)  /* string variable - output

       PF4
       IF         $SCAN (message, (24,1,length))
       THEN       rtncode = 0
       ELSE       rtncode = -1
       ENDIF
       response = line24
       ENDS
       ENDP
```

## SERINIT

**Purpose**

Initialize a communication port with the line parameters appropriate to a specific asynchronous host session.

**Format**

**[label:] SERINIT (port, speed, parity, stopbits, length, duplex, telephone_number/ machine_name, TTY_port [,flow_control])**

The **port** operand to the **SERINIT** command applies to the MS-DOS version of ESCORT and is not available in the UNIX operating system version of ESCORT. It is included for script compatibility between the UNIX operating system version and MS-DOS operating system version of ESCORT. However, a *port* number must be specified. Valid *port* numbers are 1 or 2.

**speed** specifies the speed, in bits-per-second, at which you will communicate with the asynchronous host. Valid *speeds* are:

300 bps
600 bps
1200 bps
2400 bps
4800 bps
9600 bps

**parity** specifies the type of parity setting the asynchronous host expects in the transmitted data. Valid *parity* settings are:

| | |
|---|---|
| O | Odd |
| E | Even |
| N | None |

**stopbits** specifies the number of stopbits to be transmitted, depending on whether data is transmitted at high speed. Valid *stopbits* are

either 1 or 2.

**length** specifies the length, in bits, of a transmitted data "word". Valid *lengths* are either 7 or 8.

**duplex** specifies how keyboard input is echoed to the terminal. Valid *duplex* settings are:

full
half

**telephone_number/machine_name** specifies either the telephone number or machine name of the asynchronous host. These parameters must be defined in the systems/device files associated with the UNIX operating system *uucp* facility.

**TTY_port** specifies which TTY port will be used when dialing the asynchronous host. If this parameter is not specified, a null string (" ") must be substituted in the appropriate operand position in the **SERINIT** command.

**flow_control** is an optional parameter that specifies flow control. Valid *flow_control* parameters are:

0 Disable
1 Enable

The *flow_control* parameter determines the settings of both IXON and IXOFF. The default value is 1.

Refer to the *Basic Network Utilities* documentation for further information on communication port initialization parameters.

**Remarks**

This command is effective in asynchronous sessions.

Communication port initialization parameters must be provided by a **SERINIT** statement before attempting to physically connect to an asynchronous host using the **CONNECT**

command.

The first asynchronous **CONNECT** command, in a script, establishes a connection to an asynchronous host using the parameters provided by the preceding **SERINIT** statement. Successive **CONNECT** commands to this session, reactivate the existing connection. The connection is not dropped when the script is connected to another host session. Use the **DISCON** command to disconnect sessions.

New communication port initialization parameters are provided by subsequent **SERINIT** statements. A **CONNECT** command to an asynchronous session, not already established, uses the parameters provided by the most recent **SERINIT** statement.

**See also** **CONNECT** and **PROG** commands, and the section, "Asynchronous Communication Port Initialization", in Chapter 2.

## Example

```
.
.
SERINIT (1,300,0,1,7,full,machine_a,"")  /* establish parameters
                                         /* for asynchronous host
                                         /* (machine a)
CONNECT (A1)                             /* connect to async host A1
                                         /* (machine a)
.
.
CONNECT (H1)                             /* connect to sync host H1,
                                         /* connection to async host A1
                                         /* dormant, not dropped
.
.
CONNECT (A1)                             /* reconnect to async host A1
                                         /* (machine a)
.
.
SERINIT (1,300,0,1,7,full,"5551234","")  /* establish parameters
                                         /* for asynchronous host
                                         /* (phone number 555-1234)
CONNECT (A2)                             /* connect to async host A2
                                         /* using new parameters;
                                         /* connections to async host A1
                                         /* and sync host H1
                                         /* dormant, not dropped
.
.
```

**Remarks** This command is effective only during script execution. When Interactive mode or Tutorial mode are entered, ESCORT displays the presentation space of the active session.

**See also** CONNECT command.

**Example** In the following example, synchronous host session H1 is active. The **SHOW** command is used to display various presentation spaces.

```
.
.
CONNECT (H1)        /* host session H1 is active
.
.
SHOW (L1)           /* local session L1 presentation space is displayed,
                    /* H1 continues to execute in background
.
.
EXIT                /* go to interactive mode, H1 presentation space displayed
.
.
SHOW (H1)           /* H1 presentation space redisplayed, acts the same as
                    /* a FRESH command
.
.
```

# SHOW

**Purpose** Display the presentation space of a particular session.

**Format** [label:] SHOW (session-id)

**session-id** specifies the session identification of the session to be displayed. Valid session identifications are:

| | |
|---|---|
| H1 | Synchronous host session 1 |
| H2 | Synchronous host session 2 |
| H3 | Synchronous host session 3 |
| H4 | Synchronous host session 4 |
| A1 | Asynchronous host session 1 |
| A2 | Asynchronous host session 2 |
| A3 | Asynchronous host session 3 |
| A4 | Asynchronous host session 4 |
| L1 | Local session 1 |
| L2 | Local session 2 |

# SWITCH

**Purpose** Executes a set of statements depending on the value of a string or integer argument.

**Format**

```
[label:]   SWITCH (var_name)
           CASE constant_1

              <code for case 1>
              .
              .
              .
           CASE constant_n

              <code for case n>

           DEFAULT

              <code for default case>

           ENDC
```

**var_name** specifies a string or integer variable whose value determines which **CASE** will execute. The variable name must not be a reserved word.

**constant_1** specifies a string or integer constant (depending on the type of *var_name*).

**Remarks** The value of *var_name* is compared with each of the case constants. If there is a match, the code following the matching constant statement is executed up to the next **CASE**. If no match is found, the **DEFAULT** case is executed. The **DEFAULT** case is required and is terminated by an **ENDC** statement.

A maximum of 50 cases are permitted per **SWITCH** statement.

**SWITCH** statements may not be nested; you are not allowed to use a **SWITCH** statement within another **SWITCH** statement.

**Example 1** This example demonstrates use of the **SWITCH** statement with a string variable.

```
CHAR (1)    str
CHAR (6)    name
INT         i
  .
  .
SWITCH      (name)
CASE "Bill"                   /* name = Bill
            i = 1
            str = "b"
CASE "John"                   /* name = John
            i = 2
            str = "j"
CASE "Peter"                  /* name = Peter
            i = 3
            str = "p"
CASE "Joe"                    /* name = Joe
            i = 4
            str = "j"
DEFAULT                       /* name not in case list
            i = 5
            str = "x"
ENDC
```

**Example 2** This example demonstrates use of the **SWITCH** statement with an integer variable.

```
CHAR (1)    str
INT         j
INT         i
  .
  .
SWITCH      (j)
CASE 2                        /* j = 2
            i = 1
            str = "b"
CASE 5                        /* j = 5
            i = 2
            str = "j"
CASE 12                       /* j = 12
            i = 3
            str = "p"
CASE 21                       /* j = 21
            i = 4
            str = "j"
DEFAULT                       /* j not in case list
            i = 5
            str = "x"
ENDC
```

# SYSREQ

**Purpose** Simulates the action of the system request key on the keyboard.

**Format** [label:] SYSREQ

**Remarks** This command is effective in synchronous sessions.

Most packet networks require use of a SYS_REQ key on the keyboard to establish connection with the network.

This key is used in an SNA/SDLC (or Packet Net) environment.

**See also** AID command.

**Example**

```
.
.
.
SYSREQ            /* invoke packet network
TEXT  "ORDC"      /* select network
ENTER
```

# TAB

**Purpose** Simulates action of the tab key on the keyboard.

**Format** [label:] TAB [(n)]

**n** specifies the number of tabs to be performed. The *n* can be an integer constant or an integer variable and must be enclosed in parentheses. The *n* can have a value between 1 and 64 inclusive. The default value for *n* is 1.

**See also** **$TAB** function and **BTAB** command.

## Example

```
TAB                          /* execute 1 tab
TEXT       "0000000414"
TAB  (2)                     /* execute 2 tabs
      .
      .
i   = 5
TAB  (i)                     /* execute 5 tabs
```

# TEXT

**Purpose** Simulates an operator entering data at a terminal.

**Format** **[label:] TEXT str_expr**

**str_expr** specifies the data you want to write. It is written at the current cursor position. The *str_expr* may contain a string expression that includes a string constant, string variable, string array element, screen field variable, or string function. The *str_expr* may also be a combination of any of the above operands separated by a concatenation operator. You must use a concatenation operator if the data exceeds one line. If you use more than one constant or variable, you must enclose the expression in parentheses.

**Remarks** If you enter a character in a protected field, you will receive an error message that the terminal is "input inhibited." The program will automatically exit to Interactive mode.

The **$TAB** function may be used to enter multiple data fields in a single **TEXT** statement. The **$TAB** function can also be concatenated.

If a string is longer than the field on the screen, the extra characters will appear in the next unprotected field.

**See also** **ASSIGN(=)** statement.

**Example 1**

```
TEXT ("USERID" + $TAB + "PASSWD" + $TAB + "GROUP")
ENTER
TAB (2)                                     /* select order menu
TEXT "X"
ENTER
CURSOR (6,10)                               /* position cursor
TEXT "XYZ Co."                              /* name of company
TAB
TEXT ("P.O.Box 24000" + $TAB + "PORTLAND" $TAB) /* address of company
TEXT ("NC" "97223")
ENTER
```

**Example 2** This example shows use of the concatenation operator to continue entry of data on multiple lines.

```
 .
 .
TEXT ("DATA IN TEXT STATEMENT MAY BE CONTINUED" +
     "ON MULTIPLE LINES AS SHOWN IN THIS EXAMPLE")
 .
 .
```

# TIMEOUT

**Purpose**

The first format sets a limit on the amount of time your terminal will wait for a response from a synchronous host.

The second format sets a limit on the amount of time your terminal will wait for user input in Interactive or Tutorial modes.

**Format**

**[label:] TIMEOUT (n)**

*or*

**[label:] TIMEOUT (n, EXIT)**

**n** specifies the number of minutes the program will wait for a response. The *n* may be a positive integer constant or a positive integer variable, in the range 1 to 60.

**EXIT** is the keyword used to indicate that the time-out period corresponds to an *EXIT* command.

**Remarks**

The first format of this command is effective in synchronous sessions. The second format of this command is effective in synchronous, asynchronous and local sessions.

If an integer value is assigned to the response time operand, *n*, and the time-out value expires during a synchronous host session, your terminal will be put in Interactive mode. This gives you an opportunity to fix any problems and restart the script. In order for you to restart the script successfully, the host sessions' status must duplicate the position at the time script execution was interrupted. Script execution can be resumed by pressing (ESC) **f 2**. If a time-out value is not set, the default value is 60 minutes.

If an integer value is assigned to the response time operand, *n* when the *EXIT* keyword is specified, and the time-out value expires in Interactive or Tutorial modes, control returns to

the script. This command is useful in ensuring that the terminal is not left in Interactive mode for an indefinite period.

The internal global integer variable, *SYSRET*, returns the result of an Interactive mode time-out operation. *SYSRET* may have one of the following values after control returns to the script following a **TIMEOUT (n, EXIT)** command:

| | |
|---|---|
| 0 | **TIMEOUT** period not expired |
| −1 | **TIMEOUT** period expired |

The specified **TIMEOUT** value remains in effect until changed by another **TIMEOUT** command.

See also **EXIT** command.

**Example**

```
 .
 .
TIMEOUT (5)          /* change sync host time-out to 5 minutes
 .
 .
TIMEOUT (15, EXIT)   /* allow user to enter data within 15 minutes
 .
 .
```

# WAIT

**Purpose**

The first format temporarily delays script execution.

The second format searches an asynchronous data stream.

**Format**

**[label:] WAIT (seconds)**

*or*

**[label:] WAIT (seconds, search-str__1 [,search-str__2] . . . [,search-str__8])**

**seconds** specifies the number of seconds script execution is suspended. The *seconds* may be an integer constant or an integer variable.

**search-str__1** to **search-str__8** specify the strings to be searched for in the incoming asynchronous host data stream. The *search-str__1* to *search-str__8* operands may contain string expressions that include string constants, string variables, string array elements, screen field variables, or string functions. If you use more than one constant or variable, you must enclose the expression in parentheses.

Any one of the eight search string operands may be substituted by the system global variable, *SYSPRMT*.

**Remarks**

When using the asynchronous format **WAIT** command, control is immediately returned to the script when one of the search strings is detected in the incoming data stream. The system global variable, *SYSRET*, returns the result of an asynchronous **WAIT** operation. *SYSRET* may have one of the following values after the **WAIT** operation:

| | |
|---|---|
| n | *search-str_n* detected (*n* is the positional number, in the **WAIT** command, of the matched search string) |
| –1 | No search string detected within time-out value specified by *seconds* |

By substituting a search string with the system global variable, *SYSPRMT*, the **WAIT** command searches for the specified asynchronous host system prompt at a specified screen location or throughout the entire data stream.

The *SYSPRMT* variable is initialized by the **PROMPT** command.

**See also** **PROMPT** command and the section,"Synchronizing Data Transmissions", in Chapter 2.

**Example 1** In this example, script execution is delayed for 20 seconds before the file is re-read to determine whether the necessary flag has been received.

```
.
.
WHILE t = 0                        /* execute loop while t is equal to 0
DO
  READ (file,var)
  IF var != "flag"                 /* flag not received
  THEN
    WAIT (20)                      /* delay before rereading file
    CYCLE
  ENDIF
.                                  /* procedure if flag received
.
ENDO
.
.
```

**Example 2** In this example, the **WAIT** command searches the data from the asynchronous host for either a specific response or for the dollar sign system prompt in column 1.

```
.
.
PROMPT ("$",1)
.
.
CONNECT (A1)
.
.       (transactions with async host)
.
WAIT (10,"Update Successful",SYSPRMT)
IF SYSRET = 1
THEN
  PRINT ("Update Successful")
ELSE
  IF SYSRET = 2
  THEN
    PRINT ("System Prompt Found")
  ELSE
    PRINT ("Timed Out")
  ENDIF
ENDIF
.
.
```

## WHILE

**Purpose** Allows repetitive execution of a block of code (*loop*) as long as a given condition is true.

**Format**

```
[label:]  WHILE  clause
          DO
            .
            .
           statement(s)
            .
            .
          ENDO
```

**clause** specifies an expression that returns a *true* (non-zero) or a *false* (zero) value. The expression can be an integer expression, a relational expression, or a combination of these expressions separated by "&" or "|" operators. A relational expression always returns an integer value (zero for *false*, non-zero for *true* condition).

**statement(s)** specifies a block of ESCORT code.

**Remarks** As long as the *clause* returns a true value, the statements between the DO and ENDO are executed repeatedly.

At least one statement is required between the DO and ENDO. DO and ENDO are always required. Labels are not allowed on DO and ENDO.

Nested **WHILE** statements are allowed.

A $SCAN function can be used to search for a given string in the screen buffer.

The **BREAK** and **CYCLE** commands may be used between DO and ENDO.

**See also** **BREAK** and **CYCLE** commands and **$SCAN** function.

**Example 1** This example demonstrates use of nested **WHILE** loops.

```
CHAR(80) REC(5)
INT i
INT j
i   =  25
WHILE  i != 0                  /* execute loop while i is not equal to 0
DO
       WHILE j < 10            /* nested WHILE
       DO
             j = (j+1)         /* delay
       ENDO
          .
          .
       CALL ADCUST            /* subroutine to add a customer
          .
          .
       i  = (i-1)
ENDO
```

**Example 2** This example demonstrates use of a **WHILE** loop with string and relational expressions.

```
WHILE (string1 = "abcdef")      /* string variable/constant
           |                   /* or
      (counter1 = counter2)     /* integer variables
           |                   /* or
      ($SCAN ("ADD COMPLETED")) /* field scan
DO
      .
      .
      IF    a = b
      THEN  BREAK               /* branches after ENDO
      ELSE
        .
        .
      ENDIF
      .
      .
ENDO
```

# WINDOW

**Purpose** Defines a rectangular area on your screen in which you can write messages to a terminal operator. The **WTO** command is used to write messages in the window.

**Format** [label:] WINDOW (r1,c1, r2,c2 [,R])

**r1,c1** defines the row and column address of the top left corner of the window. The row and column operands can be either integer constants or integer variables.

**r2,c2** defines the row and column address of the bottom right corner of the window. The row and column operands can be either integer constants or integer variables.

**R** is an optional parameter that defines the window as a *resident* window. A resident window stays on your screen after the arrival of a new message from the host. If a window is not defined as resident, it will disappear when a new message arrives from the host.

**Remarks** Each window has a border and therefore must span at least 3 rows and 3 columns. The maximum window size is 24x80 characters (the entire screen).

When the **WINDOW** command is executed, a window is drawn on your screen which contains no data.

The window temporarily covers the application display in the defined area. If you erase a resident window by using the **ERASEW** command, you will see the contents of the screen buffer underneath the window.

The **WTO** command is used to write messages in the window. ESCORT performs word-spill processing at the end of a window line. Words cannot be split between lines. When an entire

word does not fit on a line, it is moved to the next.

Each new message starts on a new line in the window. When the window is full, the message scrolls up one line.

You may clear the contents of a window by coding **WTO** " " for each line in the window or by issuing the **WINDOW** command again using the same row and column values as before.

If you define a new window before an old window is erased, the screen will contain multiple windows. However, you can only write to the last window. The **ERASEW** command erases all existing windows. Any resident window is also eliminated by **ERASEW**.

If no window is active, any **WTO** message is written to a default area, the operator information area.

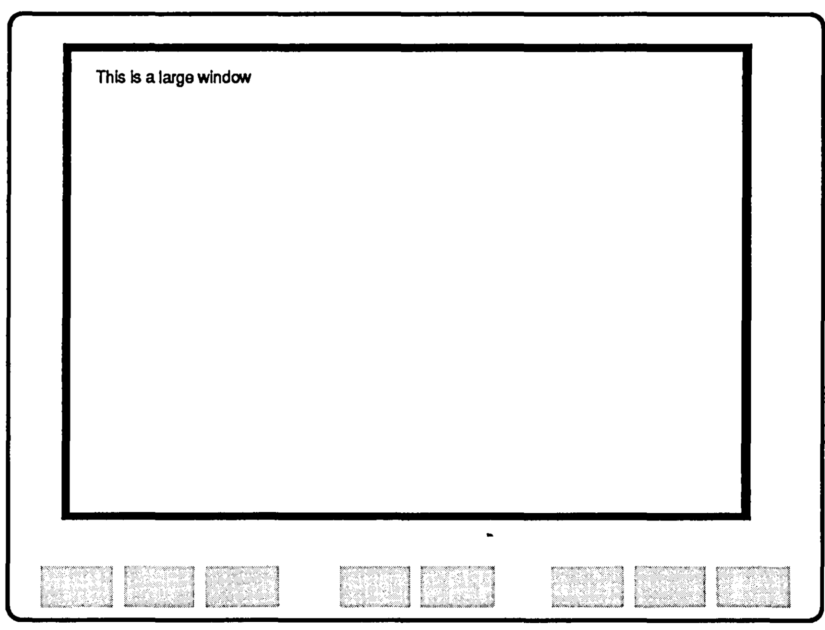
**See also** **ERASEW** and **WTO** commands.

**Example 1** This example shows you how to use integer variables to define a non-resident window.

```
rl  =  9
cl  =  60
r2  =  12
c2  =  77
WINDOW (rl,cl,r2,c2)  /* define window using variables
WTO    "Press enter key to continue."
EXIT   (TUTORIAL)
```

**Example 2** This example shows you how to use integer constants to define a non-resident window.

```
WINDOW (2,5,22,75)    /* define non-resident window
WTO    "This is a large window."
```




**Example 3** This example shows you how to create another resident window.

```
WINDOW (4,4,8,42,R)   /* define resident window
WTO    ("This is a small window." +
        "It is a resident window.")
/**employs word-spill
EXIT   (TUTORIAL)
CLEAR                 /* clear screen but resident window stays
WTO    "Enter 'Imstest'.  Then press enter key."
EXIT   (TUTORIAL)
```

# WRITE

**Purpose** Writes a record to a file.

**Format** **[label:] WRITE (nickname, buffer)**

**nickname** specifies the internal name of the file. This is the name of the file you assigned in the **OPEN** statement. The *nickname* is global and can be used in any script within the entire program.

**buffer** specifies the symbolic name of a string variable from which the data record is written. The *buffer* size must be equal to the maximum record size in the file (the maximum possible record size is 2048 characters).

**Remarks** You must open a file for write or append mode before attempting to write to it.

The **WRITE** operation is a sequential operation. Each **WRITE** operation writes data in the file at the end of the last record.

Data is not written from the internal system buffer to the file unless the internal system buffer is full or a **CHKPT** command in a script is encountered. In the event of a system failure, data in the internal system buffer is lost. If data is critical, therefore, a **CHKPT** command should be performed after each **WRITE** command. Such frequent use of the **CHKPT** command may cause slight degradation in script performance.

Checking for a successful **WRITE** operation is good programming practice. The internal global integer variable, *SYSRET*, returns the result of a **WRITE** operation. *SYSRET* may have one of the following values after the **WRITE** is executed:

| | |
|---|---|
| 0 | Successful **WRITE** |
| −1 | Failed **WRITE** |

If a file is opened as a pipe between scripts, it will be necessary to establish an end-of-file flag that will be recognized by the script reading data from the pipe, since the end-of-file condition returned by *SYSRET* may be ignored in the reading script. Refer to the "Writing to a Pipe File" script in Appendix G for an example.

See also

**CHKPT, CLOSE, OPEN, READ,** and **WAIT** commands.

Example

In the following example, records are written sequentially to a file (nickname *F1*) from an array. The file contains variable length records. The maximum record size in this file is 80 bytes. For each record, the program prints out sequence number, length, and contents.

```
CHAR (80)  rec (25)       /* 80 byte record array
CHAR (80)  buffer         /* 80 byte buffer
INT        length
INT        record

OPEN (F1, "FILE1", W)     /* open file for write
OPEN (F2, "FILE2", R)     /* open file for read

FOR record = 1 to 25
DO                        /* execute following code
  READ     (F2, buffer)   /* read next record
  length = $LENGTH(buffer)
  PRINT    ("Record # " + $ITOS(record))
  PRINT    ("Length = " + $ITOS(length))
  WRITE    (F1, buffer)
  IF  SYSRET = -1 THEN
           ABEND (12)     /* user abend code
  ENDIF
ENDO
```

# WTO

**Purpose** Writes a message to the operator, on the screen.

**Format** [label:] **WTO str_expr**

**str_expr** specifies the data to be written in a window or to the operator information area. The *str_expr* may contain a string expression that includes a string constant, string variable, screen field variable, string array element, or string function. It may also be a combination of any of the above operands separated by a concatenation operator. If you use more than one constant or variable, you must enclose the expression in parentheses.

**Remarks** The **WTO** (Write To Operator) command may be used to communicate with the terminal operator.

If a window is active, the message is displayed in the window, beginning at the next free line. Successive messages may be sent to the same window. If there is insufficient space left in the window to complete the message, the window scrolls up.

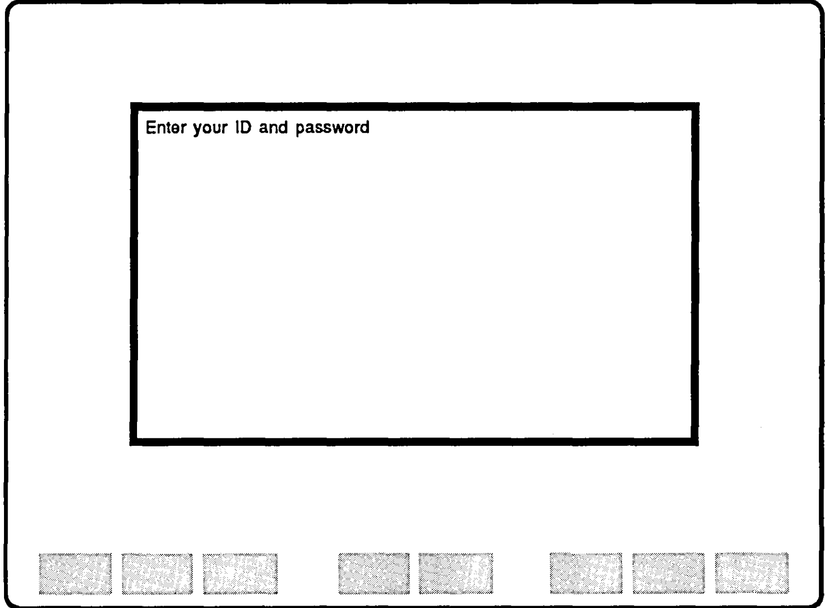
Each **WTO** message starts on a new line inside the window. If a word cannot fit at the end of the line, word spill processing occurs and the word is written on the next line.

If no window is active, the message is displayed in the operator information area. If data exceeds 60 characters and no window is active, excess characters are lost.

**See also** **ERASEW** and **WINDOW** commands.

**Example 1** This and the next example show how to use the **WTO** command with a string constant.

```
  .
  .
WINDOW (5,10,20,70)
WTO    "Enter your ID and password."
  .
  .
```



## Example 2

```
INT     i
CHAR (1) codetype (4)
codetype = ("a", "b", "c", "d")    /* initialize array
   .
   .
   .
TEXT  "/for logon"
ENTER                       /* get logon screen
WTO   "ENTER USERID AND PASSWORDS THEN PRESS ESC f 2"
EXIT                        /* enter interactive mode

/* program continues here after ESC f 2 pressed
ENTER
WHILE $SCAN ("LOGON FAILED" (24,1,80))
DO
      WTO   "TRY AGAIN"
      EXIT                          /* enter interactive mode
      ENTER
ENDO
```

## Example 3

This example demonstrates use of the **WTO** command with functions and a string constant.

```
WTO   ($DATE + "  " $TIME + " TEST002 COMPLETED SUCCESSFULLY")
BEEP
```

## Example 4

This example demonstrates use of the **WTO** command with a string array.

```
WINDOW (10,15,19,30)              /* define window
WTO    "Below is a list of valid code types:"
FOR    i = 1 to 4
DO                                /* write to successive window lines
       WTO ("code " $ITOS(i) "=" codetype (i))
ENDO
```

# Function Directory

This function directory contains a complete alphabetical listing of all ESCORT functions.

## Conventions Used

Most functions have the following format:

**$FUNCTION operands**

Function names always start with a dollar sign ($) and are printed in capital letters.

Optional fields are noted in brackets. In the example below, you may enter a number (*n*) to indicate the number of tabs to be executed.

**$TAB [(n)]**

Braces indicate a choice of operands. In the example below, you must enter one of the listed operands (*P*, *A*, *H*, *D*, or *M*).

```
$ATTR    (position,    {P})
                       {A}
                       {H}
                       {D}
                       {M}
```

Operands are separated by commas, as in the example below:

**$CHDATE (n, date)**

Parentheses must be entered where indicated. In the example above, the entries for *n* and *date* must be enclosed in parentheses.

Functions are either string or integer functions. The one exception is the **$TAB** function, which does not return a value and is therefore neither a string nor an integer function.

*String functions* may be assigned to a string variable, screen field variable, or string array element and may appear in a string or relational expression. A string function returns a character string.

*Integer functions* may be assigned to integer variables or integer array elements and may appear in an integer or relational expression. An integer function returns an integer value.

No function may appear as an operand of another function.

Several function descriptions use the term *position* or *screen position*. This term denotes the absolute position (offset + 1) rather than the row and column. This is the position returned by the **$SCAN, $FLDADDR,** and **$NEXTFLD** functions. It may also be an operand in the **CURSOR** statement. For example, row 1, column 1 is screen position 1; row 24, column 80 is screen position 1920.

When used in this section, the term *null string* means a string of length zero (no characters).

The format *mm-dd-yy* stands for month-day-year.

The format *hh:mm:ss* stands for hours:minutes:seconds.

Most examples listed in this directory are program sections. Many examples use a dot (.) on a line by itself to denote additional code.

## Function Summary

In the following table, a bullet ( ● ) indicates the session type, (synchronous, asynchronous or local) in which each function is effective.

| Command | Synchronous Host | Asynchronous Host | Local |
|---|---|---|---|
| $ATTR | ● | | ● |
| $CHDATE | ● | ● | ● |
| $DATE | ● | ● | ● |
| $DATES | ● | ● | ● |
| $DAY | ● | ● | ● |
| $EVAL | ● | ● | ● |
| $FLDADDR | ● | ● | ● |
| $GETCUR | ● | ● | ● |
| $GETENV | ● | ● | ● |
| $GETPID | ● | ● | ● |
| $GSUBSTR | ● | ● | ● |
| $HEX | ● | ● | ● |
| $ITOS | ● | ● | ● |
| $LENGTH | ● | ● | ● |
| $MONTH | ● | ● | ● |
| $NEXTFLD | ● | | ● |
| $RESP | ● | | |

| Command | Synchronous Host | Asynchronous Host | Local |
|---|---|---|---|
| $SCAN | ● | ● | ● |
| $SEC2TIM | ● | ● | ● |
| $STOI | ● | ● | ● |
| $STRIP | ● | ● | ● |
| $TAB | ● | ● | ● |
| $TIMDIFF | ● | ● | ● |
| $TIME | ● | ● | ● |
| $TIM2SEC | ● | ● | ● |
| $YEAR | ⊖ | ⊖ | ⊖ |

# $ATTR

**Purpose**

Tests an attribute of a field starting at a given position, and returns a true (non-zero) or false (zero) condition.

**Format**

$ATTR (position, {P})
{A}
{H}
{D}
{M}

**position** specifies the screen position of a field. This is the position returned by the **$FLDADDR, $GETCUR**, and **$SCAN** functions. The position is expressed as either an integer variable or a constant (in the range of 1-1920).

**attribute** specifies the mask for the attribute to be tested. Listed below are the keyword codes for the attributes that can be tested:

| Mask | True | False |
|---|---|---|
| P | Protected | Unprotected |
| A | Alphanumeric | Numeric |
| H | Highlighted | Normal |
| D | Dark | Displayable |
| M | Modified (tagged) | Not modified |

**Remarks**

This function is effective in synchronous and local sessions.

$ATTR is an integer function.

If you test a field for a specific attribute, the $ATTR function returns a *true* or *false* value, depending on whether the attribute was found.

The following is a list of the values:

| | |
|---|---|
| 1 | True - Attribute *found.* |
| 0 | False - Attribute *not found.* |
| −1 | Specified position not at start of a screen field. |

**See also** **$FLDADDR, $GETCUR,** and **$SCAN** functions.

**Example** The following example demonstrates how to use the **$ATTR** function. The program first obtains the field position of the field *duedate* and then tests it to see if it is a protected field. An error message is written if the field does not start at the given position (i.e., if there is no preceding attribute byte).

```
FIELD (5,8,6)  duedate
INT   n

n=$FLDADDR (duedate)         /* get defined field position
IF  $ATTR (n,P) = 1 THEN     /* tests for protected field
    PRINT "'duedate' is a protected field"
ELSE
IF  $ATTR (n,P) = 0 THEN     /* tests for unprotected field
    PRINT "'duedate' is an unprotected field"
ELSE
    PRINT "'duedate' does not start at defined location"
ENDIF
ENDIF
```

# $CHDATE

**Purpose** Adds or subtracts a specified number of days to a date and returns a new date.

**Format** **$CHDATE (n, date)**

**n** specifies the number of days added (or subtracted) from a given date. The *n* is a positive or negative integer constant or an integer variable.

**date** specifies the initial date to which you either add or subtract a number of days, *n*. The *date* can be either a string constant or a string variable. The *date* must be six characters long in the format *mmddyy*.

**Remarks** $CHDATE is a string function.

The receiving variable must be at least six characters long.

## Example

```
CHAR (6)  todate
CHAR (6)  duedate
INT       n
     .
     .
     .
todate  = $DATES              /* get current date
n       = 60                  /* 60 days to be added
duedate = $CHDATE (n,todate)  /* get new date
PRINT     ("Today's date = " + todate +
          ", Due Date = " + duedate)
```

# $DATE

**Purpose** Returns the current date in the format *mm-dd-yy*.

**Format** $DATE

**Remarks** $DATE is a string function.

The receiving variable must be at least eight characters long.

**See also** $DATES function.

## Example 1

```
PRINT  ("Today's date is "  + $DATE)
```

## Example 2

```
CHAR (8)        todate
CHAR (8)        dates (10)          /*string array
FIELD (5,10,8)  screena.date
 .
 .
todate     =   $DATE                /*string variable assignment
 .
dates(5)   =   $DATE                /*string array element assignment
 .
screena.date = $DATE                /*screen field variable assignment
 .
PRINT  ("DATE= " + $DATE +          /*use in string expression
       ", TIME = " + $TIME)
```

# $DATES

**Purpose** Returns the current date in the format *mmddyy*.

**Format** $DATES

**Remarks** $DATES is a string function.

The receiving variable must be at least six characters long.

**See also** $DATE function.

## Example 1

```
FIELD  (10,21,6) todate
todate = $DATES                          /* write current date to field "todate"
```

## Example 2

```
CHAR (6)        todate
CHAR (6)        dates (10)          /*string array
FIELD (5,10,6)  screena.date
 .
 .
todate    =   $DATES                 /*string variable assignment
 .
dates(5)  =   $DATES                 /*string array element assignment
 .
screena.date = $DATES                /*screen field variable assignment
 .
PRINT  ("DATE= " + $DATES +          /*use in string expression
        ", TIME = " + $TIME)
```

# $DAY

**Purpose** Returns the current day of the month in the format *dd*.

**Format** $DAY

**Remarks** $DAY is a string function.

The receiving variable must be at least two characters long.

## Example 1

```
FIELD (10,21,2) today
today = $DAY                    /* write current day to field "today"
```

## Example 2

```
CHAR (2)          day
CHAR (2)          month
CHAR (2)          year
CHAR (2)          dates (10)      /* string array
FIELD (5,11,2)    order.day
FIELD (5,14,2)    order.month
FIELD (5,17,2)    order.year
   .
day    =    $DAY                  /* string variable assignment
month  =    $MONTH                /* string variable assignment
year   =    $YEAR                 /* string variable assignment
   .
dates(5) =  $DAY                  /* string array element assignment
   .
FORMAT  order
 .day    =  $DAY                  /* screen field assignment
 .month  =  $MONTH                /* screen field assignment
 .year   =  $YEAR                 /* screen field assignment
 .
PRINT  ("DUE DATE " + $DAY +     /* use in string expression
        "_" + $MONTH + "_" + $YEAR)
```

# $EVAL

**Purpose** Performs standard arithmetic calculations on, or compares, two given numeric string operands.

**Format** **$EVAL (operand_1, operator, operand_2 [,dec])**

**operand_1** specifies the first operand in the calculation corresponding to the addend in addition, minuend in subtraction, multiplicand in multiplication, or dividend in division.

**operator** specifies the operator type. Valid operator types are:

| | |
|---|---|
| + | addition |
| − | subtraction |
| * | multiplication |
| / | division |
| % | modulo (remainder division) |
| ? | comparison |

**operand_2** specifies the second operand in the calculation corresponding to the addend in addition, subtrahend in subtraction, multiplier in multiplication, or divisor in division.

**dec** specifies the number of decimal places returned in the result. The *dec* can be an integer constant or integer variable in the range 0 to 12. The *dec* is an optional operand and, if omitted, **$EVAL** returns a default value of zero decimal places in the result.

The *operand_1* and *operand_2* must be string constants or string variables containing integer numbers or floating point numbers. The *operand_1* and *operand_2* may contain commas or leading dollar signs ( $ ). **$EVAL** ignores commas and leading dollar signs when evaluating operands.

The *dec* is ignored in comparison operations.

**Remarks**

$EVAL is a string function. The result of a $EVAL arithmetic calculation can be assigned to a character string, the length of which determines the length of the $EVAL result string. The maximum character string returned by $EVAL is 14 characters made up as follows:

| | |
|---|---|
| sign | 1 character |
| decimal point | 1 character |
| characteristic (integer) and mantissa (decimal fraction) | 12 characters |

$EVAL will truncate least significant digits from results that are longer than 14 characters.

The comparison operator returns one of three values:

| | |
|---|---|
| "−1" | *operand_1* < *operand_2* |
| "0" | *operand_1* = *operand_2* |
| "1" | *operand_1* > *operand_2* |

**Example** This example shows how to use both integer and floating point numbers and a combination thereof in **$EVAL** arithmetic calculations.

```
CHAR (3)   op_1
CHAR (10)  op_2
CHAR (10)  result_1
CHAR (14)  result_2
CHAR (2)   result_3
.
.
result_1 = $EVAL("7",+,"5")          /* result_1 = "12"
result_1 = $EVAL("7,350",-,"5,675")/* result_1 = "1675"
result_1 = $EVAL("3",*,"5.2",7)     /* result_1 = "15.6000000"
result_2 = $EVAL("$22",/,"7",12)    /* result_2 = "3.142857142857"
.
.
op_1 = "27"
op_2 = "3.12"
result_2 = $EVAL(op_1,/,op_2,12)    /* result_2 = "8.653846153846"
result_2 = $EVAL(op_1,%,op_2,12)    /* result_2 = "2.040000000000"
.
.
op_1 = "112"
op_2 = "53.6"
result_3 = $EVAL(op_1,?,op_2)       /* result_3 = "1"
result_3 = $EVAL("15.3",?,op_2)     /* result_3 = "-1"
.
.
IF $EVAL(op_1,?,op_2) = "0"         /* checks if op_1 = op_2
  THEN
  .
  .
ENDIF
.
.
IF $EVAL(op_1,+,op_2) = $ITOS(c)    /* compares result to an integer
  THEN
  .
  .
ENDIF
.
.
```

# $FLDADDR

**Purpose** Returns the absolute screen position from a given screen field.

**Format** **$FLDADDR (field_name)**

**field_name** specifies the screen field name for which you are seeking an address.

**Remarks** **$FLDADDR** is an integer function.

**See also** **$ATTR** function.

**Example** This example shows how to use **$FLDADDR** to determine at which of three fields the cursor is positioned.

```
FIELD (5,6,2) a
FIELD (7,6,2) b
FIELD (9,6,2) c
INT   i

TAB
i     =  $GETCUR              /* get cursor position
IF    $FLDADDR (a) = i THEN   /* cursor at field a
      CALL fielda             /* call subroutine
ENDIF
IF    $FLDADDR (b) = i THEN   /* cursor at field b
      CALL fieldb             /* call subroutine
ENDIF
IF    $FLDADDR (c) = i THEN   /* cursor at field c
      CALL fieldc             /* call subroutine
ENDIF
```

# $GETCUR

**Purpose** Returns the current cursor position.

**Format** $GETCUR

**Remarks** $GETCUR is an integer function.

**Example** This example shows how to read the current cursor position and convert it to a row and column address.

```
INT     r                       /* row
INT     c                       /* column
INT     offset

offset  =  ($GETCUR - 1)        /* get screen offset of cursor
r       =  (offset/80 + 1)      /* get row
c       =  (offset%80 + 1)      /* get column

WTO    ("ROW = " $ITOS(r) ", COL = "  $ITOS(c))
```

# $GETENV

**Purpose** Returns the value of a UNIX operating system environment variable.

**Format** $GETENV (evar)

**evar** specifies the environment variable whose value you wish to obtain. It must be a string constant.

**Remarks** $GETENV is a string function. The receiving variable may be up to 255 characters long.

**See also** **PUTENV** command.

**Example** This example shows how to use **$GETENV** to obtain the value of the UNIX operating system environment variable *TERM*.

```
CHAR (8) termtype
termtype = $GETENV ("TERM")
```

# $GETPID

**Purpose** Returns the process identification (PID) of ESCORT.

**Format** $GETPID

**Remarks** $GETPID is a string function. The PID has five digits.

**Example** This example shows how to use **$GETPID** to receive a PID.

```
CHAR (6) pid
CHAR (20) fname

fname = ("custf" + $GETPID)  /* make filename unique
open (file1,fname,W)
   .
   .
   .
```

# $GSUBSTR

**Purpose** Returns a substring from within a string or screen field variable.

**Format** **$GSUBSTR ({string}, position, length)**
**{SCREEN}**

**string** specifies the string or screen field variable name containing the substring you want to get.

**SCREEN** specifies the screen buffer.

**position** specifies the starting location of the substring within the *string*. The *position* can be either an integer constant or an integer variable and must have a value between 1 and the length of the entire string. If the position specified in this field is not within the range of your string, an error message will appear.

**length** specifies the number of characters the substring contains. The *length* can be either an integer constant or an integer variable and must have a value between 1 and 256. The receiving variable must be at least equal to the value specified in this field, or the substring will be truncated.

**Remarks** **$GSUBSTR** is a string function.

**Example 1** This example writes on the screen the text *I am not a string.*

```
sl = "I am a string"
TEXT ($GSUBSTR(sl,1,4) + " not a " + $GSUBSTR(sl,8,6))
```

**Example 2** This example returns three characters from the screen.

```
  .
  .
TAB                        /* position cursor
i = $GETCUR                 /* get cursor position
data = $GSUBSTR (SCREEN,i,3)
```

**Example 3** This example reads the current time of day and sends a message to the operator based on the time of day.

```
  .
  .
INT        i
INT        ti
CHAR (8)   t0
CHAR (2)   t1
CHAR (10) mea (3)              /* 3 element array

mea (1)   = "MORNING"          /* initialize the array
mea (2)   = "EVENING"
mea (3)   = "AFTERNOON"

t0  =  $TIME                   /* get current time of day
t1  =  $GSUBSTR (t0,1,2)       /* get hh from time string
ti  =  $STOI (t1)              /* convert to integer
IF     ti < 12
THEN   i = 1                   /* morning before 12 noon
ELSE   IF  ti > 17
       THEN  i = 2             /* evening after 5pm
       ELSE  i = 3             /* afternoon between 12am-5pm
       ENDIF
ENDIF
WTO    ("GOOD " + mea(i))      /* write good day message
```

# $HEX

**Purpose** Returns the ASCII character for a given hexadecimal value.

**Format** **$HEX (hexval)**

**hexval** specifies the hexadecimal value of the ASCII character you wish to obtain.

**Remarks** **$HEX** is an string function. The **$HEX** function provides you with a method of producing ASCII characters that would otherwise be interpreted with a different meaning.

**Example** Refer to the sample script, "Asynchronous Host Soft Function Keys", in Chapter 5, for an example of how to use the **$HEX** function to send data strings to an asynchronous host.

This example shows how to use **$HEX** to embed double quotation marks (") in a string. Hex 22 is the hexadecimal equivalent of the ASCII double quotation mark. Double quotation marks would otherwise mark the beginning and end of the string.

The **TEXT** command in this example will produce:

**Text with "Quotes" embedded.**

```
CHAR (30)  a
.
.
a = ("Text with " + $HEX("22") + "Quotes" +
    $HEX("22") + " embedded.")
TEXT a
```

# $ITOS

**Purpose** Converts an integer to a string.

**Format** **$ITOS (int)**

**int** specifies the integer you want to convert to a string. The integer can be a constant or a variable. It may have any value between $-2^{31}+1$ and $+2^{31}-1$, inclusive.

**Remarks** **$ITOS** is an string function.

The **$ITOS** function may be used to convert the result of an arithmetic calculation to a string.

Negative numbers are preceded by a minus sign (−). There are no leading zeros.

**See also** **$STOI** function.

## Example

```
INT             i
INT             k
CHAR (6)        s1
CHAR (6)        s2
CHAR (6)        cur
CHAR (6)        sa (10)            /* string array
FIELD (5,10,5) ord.orderno

k   =  -32767
s1  =  $ITOS(k)                     /* string -32767
k   =  256
s2  =  $ITOS(k)                     /* string 256

cur =  $GETCUR                      /* get current cursor address
sa(5) = $ITOS(cur)                  /* string array assignment
```

# $LENGTH

**Purpose** Returns the current length of a specified string variable.

**Format** **$LENGTH (str__var)**

**str__var** specifies the string variable, string array element, or screen field variable whose length you want to know.

**Remarks** **$LENGTH** is an integer function.

The length of a string variable is the length of the last string assigned. See example 1.

The length of a field is always the defined length. See example 2.

## Example 1

```
INT   j
CHAR  (10) x
x  =  "abc"
j  =  $LENGTH (x)            /* j is 3
```

## Example 2

```
FIELD (1,1,10) y
y  =  "abc"
j  =  $LENGTH (y)            /* j is 10
```

**Example 3** This is a script to find a particular substring in a given string. If the substring is found, a zero is returned in *code*. Otherwise, −1 is returned.

```
ss   SCRIPT   (CHAR (*) str,       /* source string
               CHAR (*) sub,       /* search string
               INT      code)      /* return code

INT   i
INT   n1
INT   n2

n1 = $LENGTH  (str)                /* source string length
n2 = $LENGTH  (sub)                /* search string length

FOR  i = 1 TO n1
DO
     IF  ($GSUBSTR (str,i,n2) = sub)
     THEN  code = 0                /* string found
     RETURN
     ENDIF
ENDO
code = -1                          /* string not found
ENDS
```

# $MONTH

**Purpose** Returns the current month of the year in the format *mm*.

**Format** $MONTH

**Remarks** $MONTH is a string function.

The receiving variable may be a string variable, string array element, or a screen field variable and must be at least two characters long.

The month is returned in a two-digit format (for example, 01, 02, ...11, 12).

**Example 1**

```
PRINT ("DATE:"  $MONTH  "/" $DAY "/" $YEAR)
```

**Example 2** In this example, the day, month, and year are sent to the host, and are subsequently printed out.

```
CHAR (2)        day
CHAR (2)        month
CHAR (2)        year
CHAR (2)        dates (10)       /* string array
FIELD (5,11,2)  order.day
FIELD (5,14,2)  order.month
FIELD (5,17,2)  order.year
  .
  .
  .
day       =   $DAY               /* string variable assignment
month     =   $MONTH             /* string variable assignment
year      =   $YEAR              /* string variable assignment
  .
  .
dates (2) = $MONTH               /* string array element assignment
  .
  .
enter
FORMAT     order
 .day   =   $DAY                 /* screen field assignment
 .month=    $MONTH               /* screen field assignment
 .year =    $YEAR                /* screen field assignment
  .
  .
PRINT  ("DUE DATE = " + $DAY     /* use in a string expression
        "-" + $MONTH + "-" + $YEAR)
```

# $NEXTFLD

Purpose — Returns the screen position of the next field after a given position.

Format — **$NEXTFLD (position)**

**position** specifies the screen position at which to start looking for the next field.

Remarks — This function is effective in synchronous and local sessions.

$NEXTFLD is an integer function.

If the given position holds an attribute byte, the screen position returned is the position immediately following the attribute byte.

The next field may be protected, unprotected, or a dark field.

If there are no fields following the given position, $NEXTFLD returns a zero value. A zero value is also returned if the screen is unformatted (contains no attribute characters).

If the last position on the screen (1920) is an attribute character, an attempt to find the next field returns a value of 1921.

See also — $ATTR function.

**Example 1** In this example, a screen has three fields at positions 162, 242, 322 (rows 3,4,5). The attribute byte begins each row at 161, 241, 321.

```
K = $NEXTFLD (1)          /* K = 162
K = $NEXTFLD (161)        /* K = 162
K = $NEXTFLD (162)        /* K = 242
K = $NEXTFLD (322)        /* K = 0
```

**Example 2** This example prints out the starting position of every field on a given screen. It also reports if the screen is unformatted (contains no attribute characters) and if there is an attribute character at the last position on the screen (1920).

```
flds   PROG    main
main   SCRIPT
       INT     i
       INT     j
       i = 1                 /* initial position
       j = 0                 /* number of fields
       WHILE  ((i != 0) & (i != 1921))
       DO
              j = (j + 1)
              i = $NEXTFLD(i)
              IF  (i != 0) & (i != 1921)
              THEN
                  PRINT ($ITOS(j) +
                        " at position " + $ITOS(*i))
              ENDIF
      ENDO
      IF    (i = 0) & (j = 1)
      THEN
             PRINT ("unformatted screen")
      ENDIF
      IF    (i = 1921) & (j = 1)
      THEN
             PRINT ("1 at position 1")
      ENDIF
      IF    (i = 1921)
      THEN
             PRINT ("attr. char at position 1920")
      ENDIF
      PRINT  (" ")                  /* blank line at the end
      ENDS

      ENDP
```

## $RESP

**Purpose** Returns the response time of the last transaction in hundredths of a second.

**Format** $RESP

**Remarks** This function is effective in synchronous sessions.

**$RESP** is an integer function.

ESCORT maintains two time indicators internally, which can be called *t0* and *t1*. When you press an AID key (such as ENTER, PF1, or CLEAR), the current time is recorded in *t0*. After a response is received from the host system (or the keyboard is unlocked), the time is recorded in *t1*. Whenever **$RESP** is executed, the difference between *t1* and *t0* (in hundredths of a second) is returned. This difference is the response time of the transaction.

A **FRESH** command will update *t1*. If you use a *DO/ENDO* loop containing the **FRESH** command to wait for a specific response, *t1* will contain the time the response arrived (See Example 2 below and Appendix C for sample programs that use **FRESH** in a *DO/ENDO* loop). The **FRESH** command is useful for synchronous host no-response mode transactions, when the response you are waiting for may not coincide with the unlock keyboard response. **$RESP** should be executed after the **FRESH** loop or after an AID subroutine call.

**Note:** Since the UNIX operating system provides a multi-tasking environment, the accuracy of the response time provided by **$RESP** may deteriorate as the load on the system increases.

**See also** **FRESH** command, and the "AID Subroutines Library" in Appendix C.

**Example 1** This example returns the response time after an **ENTER** command.

```
.
.
.
INT i
TEXT "ispf"
ENTER
i = $RESP
PRINT ("Response time = " + $ITOS (i))
```

**Example 2** In this example, the transaction response time is returned when the message, "ADD COMPLETED," is received from the host. It is important to limit the time consumed by the **$SCAN** function by specifying as precisely as possible the starting position and scope of the scan. ESCORT cannot detect the incoming message while the **$SCAN** function is executing.

```
TEXT "USNENJXZ"
PF1
WHILE !($SCAN ("ADD COMPLETED",(24, 1, 80)))
DO
      FRESH
ENDO
i = $RESP
PRINT ("Response = " + $ITOS(i))
```

# $SCAN

**Purpose** Searches the screen buffer for a specified string and returns its position.

**Format** **$SCAN (string [(row, col [,length])])**

**string** specifies the string you want to find in the screen buffer. The *string* can be either a string constant or a string variable. The screen buffer is referenced by a system global variable, *SCREEN*.

**row, col** specifies the starting location for the screen scan. The *row* and *col* can be either integer constants or integer variables. The default values for *row* and *col* are the first position on the screen (1,1). Entries for row and column are optional. If you do not specify a row and column, the entire screen buffer is searched.

**length** specifies the length of the search. The *length* can be either an integer constant or an integer variable. The value of *length* may be from 1 to 1920 characters (the maximum screen size). The default value is 1920. Entering a value for *length* is optional. However, you may only specify a length if the row and column are specified.

**Remarks** $SCAN is an integer function. If the search string is not found, $SCAN returns zero.

There are two main ways to use $SCAN:

- □ as a test in the clause of an **IF** or **WHILE** statement;
- □ to return a position.

The **$SCAN** function returns the position of the desired string, but does not position the cursor at the string. You must use the **CURSOR** command to position the cursor.

**See also** **$GETCUR** function and **CURSOR** command.

**Example 1** This example shows use of the **$SCAN** function with a default and specified starting position and length.

```
IF   $SCAN ("ADD COMPLETED")            /* default row, col, length
THEN
   .
   .
   .
ENDIF
IF   $SCAN ("FIND COMPLETED" (24,1,80))  /* no default
THEN
   .
   .
   .
ENDIF
```

**Example 2** This example shows use of the **$SCAN** function to search the buffer for the string "LOGON SCREEN".

```
TEXT "imstest"
ENTER
logon = "LOGON SCREEN"
WHILE   !($SCAN (logon))                 /* defaults, string variable
DO
        FRESH                            /* refreshes screen buffer
ENDO
   .
   .
   .
```

**Example 3** The following example calculates the row and column address of the string on the screen.

```
INT   r                                    /* row
INT   c                                    /* column
INT   offset
/*  find address of string
offset = ($SCAN ("000000414") - 1)
IF    offset < 0  THEN RETURN ENDIF         /* not found
r     =  (offset/80 + 1)                    /* get row
c     =  (offset%80 + 1)                    /* get column
```

**Example 4** The following example searches for a given string at a specific position on the screen and prints a message depending on whether the string is found.

```
IF ($SCAN  ("ADD COMPLETED" (24,9,13)))
THEN
    PRINT "SUCCESSFUL"
ELSE
    PRINT "UNSUCCESSFUL"
ENDIF
```

# $SEC2TIM

**Purpose** Converts time (in seconds) to time expressed as a string in the format *hh:mm:ss*.

**Format** **$SEC2TIM (int)**

**int** specifies time in seconds. The *int* can be a positive integer constant or an integer variable.

**Remarks** **$SEC2TIM** is a string function.

This function is useful in converting a time difference obtained by the **$TIMDIFF** function to a readable format.

The receiving variable must be at least eight characters long.

**See also** **$TIMDIFF** and **$TIM2SEC** functions.

**Example** This example performs 10 transactions on the host system and prints out the total time taken for all 10 transactions.

```
FOR i = 1 to 10 DO
    t1 = $TIME                          /* get start time
    ENTER                              /* enter transaction
    t2 = $TIME                          /* get end time
    el = $TIMDIFF (t1,t2)               /* calculate elapsed time
    totsec = (totsec + $TIM2SEC(el))    /* convert string to integer
                                       /* for arithmetic
ENDO
PRINT $SEC2TIM (totsec)                /* convert back to string for display
```

# $STOI

**Purpose** Converts a numeric string to an integer.

**Format** $STOI (string)

**string** specifies the string variable you want to convert to an integer. The value of the *string* must be between $-2^{31}+1$ and $+2^{31}-1$. If you exceed this range or if a nonnumeric character is found, your program will end abnormally and you will get an error message.

**Remarks** $STOI is an integer function.

Use the **$STOI** function to convert a numeric string into an integer so that arithmetic calculations can be performed.

**See also** $ITOS function.

**Example** This example captures two values, adds them together and prints out the total.

```
FIELD  (15,6,4) price
FIELD  (10,12,6) qty
INT Q
INT P
INT total
Q = $STOI (qty)
P = $STOI (price)
total = (Q * P)
PRINT  ("TOTAL = " + total)
```

# $STRIP

**Purpose** Returns a given string after removing any trailing blanks from it.

**Format** **$STRIP (str_var)**

**str_var** specifies the string or screen field variable containing trailing blanks.

**Remarks** **$STRIP** is a string function.

The *str_var* may contain up to 2048 characters.

The value of the *str_var* is not changed by using the **$STRIP** function.

**Example 1** This example removes trailing blanks from a screen field.

```
CHAR (12)       c
FIELD (2,15,12) f        /* f is a field containing "price   "
c = $STRIP (f)           /* c now contains "price"
                         /* the value of f is still "price   "
```

**Example 2** This example reads records from a file and strips any trailing blanks.

```
CHAR (80) bl             /* input buffer
 .
 .
 .
READ (fn, bl)            /* read one record
bl = $STRIP (bl)         /* strip off trailing blanks
```

# $TAB

**Purpose** Simulates the action of the tab key on the keyboard.

**Format** $TAB [(n)]

**n** specifies the number of tabs you want to execute. The *n* may be an integer constant or integer variable with a value between 1 and 64. The default value for *n* is 1.

**Remarks** The **$TAB** function is similar to the **TAB** command. However, you use the **$TAB** function within a **TEXT** command, as shown in the example below.

**$TAB** is a special function that is neither a string nor an integer function. It cannot be assigned to a variable.

**See also** **TAB** and **TEXT** command.

**Example** This example performs two tabs within a single text statement.

```
TEXT ("ABC Co."  $TAB(2)  "5632")
ENTER
```

# $TIMDIFF

**Purpose** Calculates the difference between two time strings and returns the result as a character string in the format *hh:mm:ss*.

**Format** **$TIMDIFF (time1, time2)**

**time1** specifies the earlier time. It can be either a string constant or a string variable.

**time2** specifies the later time. It can be either a string constant or a string variable.

**Remarks** **$TIMDIFF** is a string function.

The receiving variable must be at least eight characters long.

**See also** **$SEC2TIM, $TIME**, and **$TIM2SEC** functions.

## Example

```
CHAR (8) TIME1
CHAR (8) TIME2
TIME1 = $TIME
  .
  .
  .                     /* some processing
  .
  .
TIME2 = $TIME
PRINT $TIMDIFF (TIME1, TIME2)
```

## $TIME

**Purpose** Returns the current time of day in the format *hh:mm:ss*.

**Format** $TIME

**Remarks** $TIME is a string function.

The receiving variable must be at least eight characters long.

### Example

```
PRINT ("TIME = " $TIME)
```

## $TIM2SEC

**Purpose** Converts a given time string in the format *hh:mm:ss* to time expressed in seconds.

**Format** $TIM2SEC (time)

**time** specifies the time string that you want to convert to the number of seconds. The *time* can be either a string constant or a string variable in the format *hh:mm:ss*.

**Remarks** $TIM2SEC is an integer function.

**See also** $SEC2TIM function.

**Example**

```
FOR i = 1 to 10 DO
    t1 = $TIME                        /* get start time
    ENTER                            /* enter transaction
    t2 = $TIME                        /* get end time
    el = $TIMDIFF (t1,t2)             /* calculate elapsed time
    totsec = (totsec + $TIM2SEC(el))  /* convert string to integer
                                     /* for arithmetic
ENDO
PRINT $SEC2TIM (totsec)              /* convert back to string for display
```

# $YEAR

**Purpose** Returns the current year in the format *yy*.

**Format** $YEAR

**Remarks** $YEAR is a string function.

The receiving variable must be at least two characters long.

**Example**

```
PRINT ("DATE = " $MONTH "/" $DAY "/" $YEAR)
```

# 5 ESCORT Utilities


**Overview** 5-1

**Upload and Download** 5-3
Program Listing 5-7

**Generating Screen Field Variables** 5-17
Program Listing 5-20

**Get Fields** 5-25
Program Listing 5-27

**Asynchronous Host Soft Function Keys** 5-29
Program Listing 5-30

# Overview

This chapter contains information on the ESCORT utility programs that are included on your ESCORT installation diskette.

Read this chapter to learn how to

- transfer files to and from a synchronous host
- generate screen field variables for any synchronous host application screen
- read variable length records into an array
- send soft function key values to an asynchronous host.

The operation of the utilities is described and the individual program listings are provided at the end of each section.

# Upload and Download

The two scripts provided on your ESCORT installation diskette, named *upload* and *dnload*, can be used for transmitting text data files between TSO on a synchronous host computer and the 3B processor.

## Invoke Procedure

The procedure for invoking *upload* or *dnload* from the UNIX shell is described below.

1 To upload files from the 3B processor, type on the command line

   ***escort /usr/escort/slib/upload***

   To download files to the 3B processor, type on the command line

   ***escort /usr/escort/slib/dnload***

   and press RETURN.

2 The ESCORT banner screen is displayed briefly.

3 ESCORT then displays a File Transfer Facility input screen, (a local session screen defined by a local screen format). The File Transfer Facility screen indicates whether the transfer mode is upload or download.

4 You must specify source and target files and parameters in the appropriate fields. If all input fields are blank and you press RETURN the ESCORT script terminates and control returns to the UNIX shell.

   The first input field in the File Transfer Facility screen prompts

   **Enter UNIX File Name:**

You may enter the file name or the full path name for the UNIX file that is the source file in an upload or the target file in a download. Remember that the UNIX operating system is case sensitive and that the file name must be entered exactly as it appears in the directory. If the UNIX file name entered is less than 50 characters in length, press (TAB) to move the cursor to the next input field.

5 You must next enter the TSO data set name at the screen prompt

    **Enter Host DSNAME (Full Name, NO Quotes):**

   If the data set name is less than 50 characters in length, press (TAB) to move the cursor to the next input field.

6 The next File Transfer Facility screen input field is the logical record length of the file on the synchronous host system. At the prompt

    **Enter Host File's LRECL:**

   enter the logical record length; valid record lengths are between 1 and 255. If the record length entered is less than 3 characters, press (TAB) to move the cursor to the next input field.

7 The final File Transfer Facility screen input field is the record format of the synchronous host system file. At the prompt

    **Enter Host RECFM (FB or VB):**

   enter the appropriate record format, *FB* for Fixed Block or *VB* for Variable Block. The record format may be entered in either upper or lower case letters. The cursor automatically moves to the first input field, the UNIX file name.

8 Edit any of the fields as necessary, using (TAB) to skip to the next field, following the procedures in steps 4 to 7 above. When all fields have been completed correctly, press (RETURN) and the ESCORT *upload* or *dnload* script verifies the data you have entered.

   - If the UNIX file does not exist in the upload mode, or you do not have write permission in the download mode the ESCORT script responds with the error message

**Cannot open UNIX File: file_name**
**Please Re-enter UNIX File name and Press RETURN**

where *file_name* is the name of the UNIX file you entered in step 4 above. Re-enter the correct UNIX file name and press (RETURN) to continue.

- If the logical record length is incorrect, the ESCORT script displays the error message

  **Invalid LRECL - Valid Range is between 1-255**
  **Please Re-enter LRECL and Press RETURN**

  Type a valid record length and press (RETURN) to continue.

- If the record format is incorrect, the following error message is displayed:

  **Invalid RECFM - valid entries are FB or VB**
  **Please Re-enter RECFM and Press RETURN**

  Type a valid record format and press (RETURN) to continue.

- Note that the ESCORT *upload* and *dnload* scripts do not check the validity of the TSO data set name.

**9** ESCORT next displays the synchronous host application screen together with the following login prompt in a window:

**Please logon to TSO and leave at READY state,**
**then press ESC f 2 to resume script execution**

Log in to the application. You can log in to the application manually or you can use a script to log in automatically. To effect an automatic login, edit the *upload* or *dnload* scripts as appropriate to include your own login procedure. At the *Ready* state press the Resume key sequence, ( (ESC) **f 2**), to resume ESCORT script execution. The login prompt is redisplayed if you attempt to resume script execution before the login procedure is complete.

The ESCORT *upload* or *dnload* script automatically invokes IEBGENER from SYS1.LINKLIB so that no synchronous host program installation is necessary. If IEBGENER is not

contained in SYS1.LINKLIB, the user will have to modify the *upload* and *dnload* scripts (contained in /usr/escort/slib) to point to the appropriate libraries.

**10** Blocks of data being uploaded or downloaded are displayed on the terminal. The following message is displayed in a window for each screen load of *n* records transferred:

**n Records Up (Down) - loaded**

The following message, in a window, is displayed on the final screen:

**n Total Records Up (Down) - loaded**

**11** The *upload* and *dnload* scripts then redisplay the File Transfer Facility input screen. You may continue to select source and target files for transmission of data, following the procedures outlined in steps 4 to 10 above. Note that since you have already logged in to your application you do not have to repeat the procedure in step 9 above.

**12** When you have completed all uploading or downloading of data, at the File Transfer Facility screen press RETURN, leaving all input fields blank. The ESCORT script automatically logs off from the application. A count of the number of records uploaded or downloaded is written to the *escort.pr{proc-id}* file, in the directory defined by the ESCDIR environment variable. The following example indicates that 58 records were downloaded:

```
58 Total Records Down-loaded from dsname to UNIX_file_name
```

## Note

When uploading, the ESCORT script pads short records with blanks up to the logical record length. The *upload* and *dnload* scripts do not recognize tabs. Files containing special characters (for example, binary data) may not be transmitted using the *upload* or *dnload* scripts.

# Program Listing

## Downloading Files from TSO

```
/********************************************************
/*
/* This program is for downloading text files from TSO.  You
/* will be prompted for the TSO file name, the UNIX file name, and
/* the logical record length.  The host file must be cataloged.
/* The full data set name is required without quotes.
/*
/* The file to be downloaded can contain only displayable standard ASCII
/* characters.    Otherwise, transmission error may occur.
/*
/* The download is accomplished by executing IEBGENER in the
/* foreground.    A CLIST is uploaded and executed line by line to
/* run the GENER.  You may speed this process up by storing the
/* CLIST on TSO.
/*
/********************************************************

dnload  prog  main(L1)

        char (50)   unixname
        char (50)   dsname
        int          lrecl
        int          blk
        char (4)    recfm
        char (255)  buf
        char (255)  tmpbuf

        int          rtncode

        copy "parms.1"            /* local screen for Parms

main     script

         int         b
         int         e
         int         i
         int         j
         int         k
         int         l
         int         m
         int         tot
         int         consw
         int         endsw
         int         fldpos

 while(1)
 do
        call getdata              /* Get User Parameters for DNLOAD
        if (rtncode = 1)          /* No more files to DNLOAD - EXIT
        then
                 connect(H1)
                 if (sysret = -1)  /* Host connection failed
                 then
                   connect(L1)
                   fldpos = $fldaddr(parms.errmsgl)
                   chgattr(L1,fldpos,(U,*,*,*,*,*,*))
```

```
                parms.errmsgl = ("Cannot Connect to Host - Please Try Later")
                chgattr(L1,fldpos,(P,*,*,*,*,*,*))
                show(L1)
                exit
                return
              endif
              if ($scan("READY   "))
              then
                       show(H1)
                       text      "LOGOFF"  /* Log off TSO
                       enter
              endif
              return                       /* Exit DNLOAD
     endif

     /******************************************
     /* Exit to Interactive Mode to allow user to
     /* logon to TSO and bring to READY state.
     /******************************************

manlog: while !($scan("READY   ")) do
              window  (21,20,24,79)
              wto  (" Please logon to TSO and leave at READY state,")
              wto  (" then press ESC f 2 to resume script execution.")
              exit
      endo

      call allocate              /* allocate iebgener files

      e = 81                     /* initialize end of line
      tot = 0
      endsw = 0
      consw = 0
      window (22,50,24,79,r)    /* message window
      wto ("     Down-loading Data     ")

      k = 1
      while k = 1 do
          if tot != 0 then
               e = 0
          endif
          cursor (24, 80)
          for i = 0 to 23 do    /* is this last page ?
               j = (80*i + 2)
               if $gsubstr (screen, j, 5) = "READY" then
                   endsw = 1
                   j = (j + 80)
                   cursor (j)
                   break
               endif
          endo
          for i = 1 to 24 do      /*   process current page
               b = $nextfld (e)    /* get beginning of record
               if b = 1842 then    /* bottom ?
                 break
               endif
               if endsw = 1 then  /* not full page
                   if $gsubstr(screen, b, 5) = "READY" then
                        k = 2
                        break
                   endif
               endif
               e = $nextfld (b)        /* get end of record
```

```
                l = (e - b - 1)        /* get record length
                if consw = 1 then
                    tmpbuf = $gsubstr(screen,b,l)
                    if ((l + $length(buf)) > lrecl) |
                        (($length(tmpbuf) = 1) & tmpbuf = " ") then
                        call writeo     /* write out previous record
                        buf = $gsubstr(screen, b, l)
                        tot = (tot + 1)
                    else                /* concatenate records
                        buf = (buf + $gsubstr(screen, b, l) )
                    endif
                else
                    buf = $gsubstr(screen, b, l)
                endif
                consw = 0
                if (e = 1842)    then    /* rec continues next page
                    consw = 1
                    break
                endif
                call writeo              /* write a record
                if $attr(e, H) then      /* means no real end attr
                    e = (e - 1)          /* so this is begin attr
                endif                    /* for next record
                tot = (tot + 1)
            endo
            wto ("  " $itos(tot) "  Records Down-loaded")
            enter
        endo

        window (22,44,24,79)           /* non-resident message window
        wto ("  " $itos(tot) "  Total Records Down-loaded")
        print ($itos(tot) "  Total Records Down-loaded"
        " From " dsname " To " unixname)

        close(unixfile)

 endo   /* MAIN WHILE LOOP
        ends

writeo  script                         /* write a record
        write (unixfile, buf)
        if sysret = -1 then
           window (21,40,24,79)
           wto   "  Write Error - Aborted"
           wto   "  ENTER to terminate"
           exit  (tutorial)
           abend
        endif
        ends

allocate script
        clear
        text "FREE FI(SYSIN SYSPRINT SYSUT1 SYSUT2)  ATTR(L)"
        enter
        text "ALLOC FI(SYSIN) DA('NULLFILE') SHR"
        enter
        text "ALLOC FI(SYSPRINT) DA('NULLFILE') SHR"
        enter
        text ("ATTR L LRECL(" lrecl ") BLKSIZE(" blk ")" )
        enter
        text ("ALLOC FI(SYSUT1) DA('" dsname "') SHR ")
        enter
        text "ALLOC FI(SYSUT2) DA(*) USING(L)"
        enter
```

```
         clear
         text "CALL 'SYS1.LINKLIB(IEBGENER)'"
         enter
ends

getdata script
        int fldpos
        char(3) work

        rtncode = 0

        getfmt(L1,parms)
        connect(L1)
        show(L1)
        fldpos = $fldaddr(parms.process)
        chgattr(L1,fldpos,(U,*,*,*,*,*,*))
        parms.process = "DNLOAD"
        chgattr(L1,fldpos,(P,*,H,*,*,*,*))

 while(1)
 do
        exit                            /* Get User Parameters

        unixname = $strip(parms.unixname)  /* Get UNIX file name
        dsname = $strip(parms.dsname)         /* Get DSNAME
        work = $strip(parms.lrecl)         /* Get LRECL
        recfm = $strip(parms.recfm)           /* Get RECFM

        if (unixname = "" & dsname = "" & work = "" & recfm = "")
        then                                  /* Exit File transfer
                rtncode = 1
                return
        endif

        open (unixfile,unixname, W)
        if (sysret != 0)
        then
                call error(1)
                cycle
        endif

        lrecl = $stoi(work)                   /* Convert to integer
        if (lrecl < 1 | lrecl > 255)
        then
                call error(2)
                cycle
        endif

        blk = (lrecl * 10)
        switch (recfm)
        case    "fb"
        case    "FB"
        case    "vb"
                blk = (blk + 4)               /* for record desc word
        case    "VB"
                blk = (blk + 4)               /* for record desc word
        default
                call error(3)
                cycle
        endc

        break                                 /* Break out of while loop

 endo
```

```
  connect(H1)                                    /* Connect Back to Host TSO
  show(H1)

ends

error   script(int      code)
        int     fldpos

        fldpos = $fldaddr(parms.errmsg1)    /* Unprotect Error Message Field
        chgattr(L1, fldpos, (U,*,*,*,*,*,*))
        fldpos = $fldaddr(parms.errmsg2)
        chgattr(L1, fldpos, (U,*,*,*,*,*,*))

        switch(code)
        case 1
          parms.errmsg1 = ("Cannot Open UNIX File: " UNIXNAME)
          parms.errmsg2 = "Please Re-enter UNIX File name and Press RETURN"
          fldpos = $fldaddr(parms.unixname) /* Position Cursor
          cursor(fldpos)
        case 2
          parms.errmsg1 = "Invalid LRECL - Valid Range is between 1-255"
          parms.errmsg2 = "Please Re-enter LRECL and Press RETURN"
          fldpos = $fldaddr(parms.lrecl)
          cursor(fldpos)
          close(unixfile)
        case 3
          parms.errmsg1 = "Invalid RECFM - Valid Entries are FB or VB"
          parms.errmsg2 = "Please Re-enter RECFM and Press RETURN"
          fldpos = $fldaddr(parms.recfm)
          cursor(fldpos)
          close(unixfile)
        default
        endc

        fldpos = $fldaddr(parms.errmsg1)    /* Protect Error Message Field
        chgattr(L1, fldpos, (P,*,*,*,*,*,*))
        fldpos = $fldaddr(parms.errmsg2)
        chgattr(L1, fldpos, (P,*,*,*,*,*,*))

ends

endp
```

# Uploading Files to TSO

```
/**********************************************************************
/*
/* This program is for uploading text files to TSO.  You
/* will be prompted for the TSO file name, the UNIX file name, and
/* the logical record length.  The host file must be cataloged.
/* The full data set name is required without quotes.
/*
/* The file to be uploaded can contain only displayable standard ASCII
/* characters.    Otherwise, transmission error may occur.
/*
/* The upload is accomplished by executing IEBGENER in the
/* foreground.    A CLIST is uploaded and executed line by line to
/* run the GENER.  You may speed this process up by storing the
/* CLIST on TSO.
/*
/**********************************************************************

upload    prog  main(L1)

         char (50)   dsname
         char (50)   unixname
         int          lrecl
         char (4)    recfm
         int          rtncode

         copy     "/usr/escort/slib/parms.l"

main      script

          int         i
          int         j
          int         k
          int         l
          int         m
          int         lim
          int         tot
          int         count
          int         fldpos

          char (255)  buf
          char (255)  a (22)
          char (2048) block
          char (80)   blank

          blank = (" " blank)                    /* initialize with blanks

 while(1)
 do

         call getdata                        /* Get User Parameters for UPLOAD
         if (rtncode = 1)           /* No more files to UPLOAD - EXIT
         then
                  connect(H1)
                  if (sysret = -1)  /* Host connection failed
                  then
                    connect(L1)
                    fldpos = $fldaddr(parms.errmsgl)
                    chgattr(L1,fldpos,(U,*,*,*,*,*,*))
                    parms.errmsgl = ("Cannot Connect to Host - Please Try Later")
                    chgattr(L1,fldpos,(P,*,*,*,*,*,*))
```

```
          show(L1)
          exit
          return
        endif
        if ($scan("READY   "))
        then
                show(H1)
                text     "LOGOFF"  /* Log off TSO
                enter
        endif
        return                     /* Exit DNLOAD
endif

 /*******************************************/
 /* LOGON TO TSO --
 /* Exit to Interactive Mode to allow user to
 /* logon to TSO and bring to READY state.
 /*******************************************/

while !($scan("READY   ")) do
        window  (21,20,24,79)
        wto  (" Please logon to TSO and leave at READY state,")
        wto  (" then press ESC f 2 to resume script execution.")
        exit
 endo

 lim = (1760 - lrecl)              /* set bound on block size

 call allocate                     /* allocate gener files

 count = 0
 window (22,50,24,79,r)              /* message window
 wto ("      Up-loading Data     ")

 k = 1
 while k = 1 do
     tot = 0
     for i = 1 to 21 do
          read (unixfile, buf)
          if sysret = -1 then                  /* end of file ?
              k = 0
              a (i) = ""
              break
          endif
          count = (count + 1)
          l = $length (buf)
          j = (lrecl - l)
          if j < 0 then
            erasew
            window (21,40,24,79)
            wto    ("  Record read > " $itos(lrecl))
            wto    "  ENTER to terminate"
            print ("  Record read > " $itos(lrecl))
            exit  (tutorial)
            abend
          endif
          m = (j / 80)
          j = (j % 80)
          switch (m)                /* pad record with blanks
            case 0
              a (i) = (buf $gsubstr (blank, 1, j))
            case 1
              a (i) = (buf blank $gsubstr (blank, 1, j))
```

```
                case 2
                  a (i) = (buf blank blank $gsubstr (blank, 1, j))
                case 3
                  a (i) = (buf blank blank blank $gsubstr (blank, 1, j))
                default
                  abend (12)
              endc
              tot = (tot + lrecl)
              if tot > lim then                    /* max block
                  break
              endif
          endo

          for i = (i+1) to 21 do
              a (i) = ""                         /* null unused elements
          endo
          block = (a(1) a(2) a(3) a(4) a(5) a(6) a(7) a(8) a(9) a(10)
                    a(11) a(12) a(13) a(14) a(15) a(16) a(17) a(18) a(19)
                    a(20) a(21) )                   /* build block
          if $length (block) = 0 then
              break
          endif
          clear
          sysret = -5                     /* required for upload
          text block
          sysret = 0                               /* reset "sysret"
          enter
          wto ("  " $itos(count) "  Records Up-loaded  ")
      endo

      sysret = 0
      text "/*"
      enter
      erasew
      window (22,44,24,79)                     /* non-resident message window
      wto ("  " $itos(count) "  Total Records Up-loaded  ")
      print ($itos(count) "  Total Records Up-loaded"
      " From " unixname " To " dsname)

     close(unixfile)
 endo    /* MAIN WHILE LOOP

ends

allocate script

      clear
      text "FREE FI(SYSIN SYSPRINT SYSUT1 SYSUT2)  ATTR(L)"
      enter
      text "ALLOC FI(SYSIN) DA('NULLFILE') SHR"
      enter
      text "ALLOC FI(SYSPRINT) DA('NULLFILE') SHR"
      enter
      text ("ATTR L LRECL(" $itos(lrecl) ")")
      enter
      text "ALLOC FI(SYSUT1) DA(*) USING(L)"
      enter
      text ("ALLOC FI(SYSUT2) DA('" dsname "') SHR ")
      enter
      text "CALL 'SYS1.LINKLIB(IEBGENER)'"
      enter
      clear
 ends
```

```
getdata  script

         int fldpos
         char(3) work

         rtncode = 0

         getfmt(L1,parms)
         connect(L1)
         show(L1)
         fldpos = $fldaddr(parms.process)
         chgattr(L1,fldpos,(U,*,*,*,*,*,*))
         parms.process = "UPLOAD"
         chgattr(L1,fldpos,(P,*,H,*,*,*,*))

 while(1)
 do
         exit                                          /* Get User Parameters

         unixname = $strip(parms.unixname)   /* Get UNIX file name
         dsname = $strip(parms.dsname)                 /* Get DSNAME
         work = $strip(parms.lrecl)          /* Get LRECL
         recfm = $strip(parms.recfm)                   /* Get RECFM

         if (unixname = "" & dsname = "" & work = "" & recfm = "")
         then                                          /* Exit File transfer
                  rtncode = 1
                  return
         endif

         open (unixfile,unixname, R)
         if (sysret != 0)
         then
                  call error(1)
                  cycle
         endif

         lrecl = $stoi(work)                           /* Convert to integer
         if (lrecl < 1 | lrecl > 255)
         then
                  call error(2)
                  cycle
         endif

         switch (recfm)
         case     "fb"
         case     "FB"
         case     "vb"
         case     "VB"
         default
                  call error(3)
                  cycle
         endc

         break                              /* Break out of while loop

 endo

 connect(H1)                                /* Connect Back to Host TSO
 show(H1)

ends

error    script(int        code)
```

```
        int     fldpos

        fldpos = $fldaddr(parms.errmsg1)    /* Unprotect Error Message Field
        chgattr(L1, fldpos, (U,*,*,*,*,*,*))
        fldpos = $fldaddr(parms.errmsg2)
        chgattr(L1, fldpos, (U,*,*,*,*,*,*))

        switch(code)
        case 1
          parms.errmsg1 = ("Cannot Open UNIX File: " UNIXNAME)
          parms.errmsg2 = "Please Re-enter UNIX File name and Press RETURN"
          fldpos = $fldaddr(parms.unixname) /* Position Cursor
          cursor(fldpos)
        case 2
          parms.errmsg1 = "Invalid LRECL - Valid Range is between 1-255"
          parms.errmsg2 = "Please Re-enter LRECL and Press RETURN"
          fldpos = $fldaddr(parms.lrecl)
          cursor(fldpos)
          close(unixfile)
        case 3
          parms.errmsg1 = "Invalid RECFM - Valid Entries are FB or VB"
          parms.errmsg2 = "Please Re-enter RECFM and Press RETURN"
          fldpos = $fldaddr(parms.recfm)
          cursor(fldpos)
          close(unixfile)
        default
        endc

        fldpos = $fldaddr(parms.errmsg1)    /* Protect Error Message Field
        chgattr(L1, fldpos, (P,*,*,*,*,*,*))
        fldpos = $fldaddr(parms.errmsg2)
        chgattr(L1, fldpos, (P,*,*,*,*,*,*))

ends

endp
```

# Generating Screen Field Variables

The ESCORT script named *fldgen*, contained on the ESCORT installation diskette, can be used to generate screen field variables for any screen within your synchronous host application.

## Procedure for Generating Variables

The procedure for generating screen field variables follows:

1. On the command line, type

   ***escort /usr/escort/slib/fldgen filename[,ALL]***

   and press RETURN.

   *filename* is the name of the output file to which the generated screen field variables will be written.

   The optional parameter *ALL* can be used to generate screen field variables for all fields on the screen (both protected and unprotected). If you omit this parameter, field statements for unprotected fields only are generated.

2. The ESCORT banner screen is displayed briefly. ESCORT then displays the appropriate synchronous host application screen together with the following field generation prompt in a window:

   **Select Application Screen and**
   **-Press ESC f 2 to generate field variables, OR**
   **-Log off and exit ESCORT (ESC f 1) to quit**

3. Log in to the application (either manually or via a login script).

4. Select the application screen for which you want to generate screen field variables.

5 Press the Interrupt/Resume (I/R) key combination, ESC **f 2**, to generate the field variables for the application screen you have selected. The field generation prompt and window are redisplayed. The window and its contents do not affect the generation of screen field variables that may be obscured by this prompt.

6 Repeat the operations in steps 4 and 5 above for all remaining application screens for which you want to generate screen field variables.

7 When you have generated all the required screen field variables, log off from the application in the usual way and press ESC **f 1** to quit ESCORT and return to the UNIX shell.

## Generated Variables Format

The output file, *filename*, contains the screen field variables for each application screen that you selected. A blank line separates field statements for each application screen. The format of the field statement generated in the output file is

**FIELD(row,col,length)fld{n}/*Attributes, Groups 1 to 4 (offset)**

**row,col,length** follow the conventions defined in the **FIELD** statement in Chapter 4.

**fld{n}** specifies the field name automatically assigned by the *fldgen* script, where *n* is a sequential number starting at 0001. The first **FIELD** statement generated by *fldgen* from each application screen is assigned the field name *fld0001*; subsequent fields are named *fld0002*, *fld0003*, etc.

It is recommended that the field names in the output file be amended to unique names to avoid conflicts in field names across multiple screen definitions.

**/*Attributes, Groups 1 to 4** specifies the Primary Attributes - Group 1 to Group 4 for the generated field. The attributes are shown as a comment to the **FIELD** statement. Note that *fldgen* does not generate comments for Extended Field Attributes - Group 5 to Group 7.

The following table lists the comments generated by the *fldgen* script together with their attribute group and meaning.

| Attribute Comment | Attribute | Primary Attribute |
|---|---|---|
| PROT | Protected | Group 1 |
| UNPR | Unprotected | Group 1 |
| NUMR | Numeric | Group 2 |
| ALPH | Alphabetic | Group 2 |
| NORM DISP | Normal | Group 3 |
| HILT DISP | Highlighted | Group 3 |
| DARK | Dark | Group 3 |
| TAGS | Modified DT | Group 4 |
| TAGR | Reset DT | Group 4 |

To use the output file **FIELD** statements to generate a local screen format containing Primary Attributes and Extended Field Attributes, you must amend such **FIELD** statements by replacing the attribute comments with the corresponding attribute operand. See the **FIELD** statement in Chapter 4 for more information on Primary Attributes and Extended Field Attributes.

**(offset)** specifies the absolute screen address of the first character of the field generated. This comment may be deleted when you use the generated **FIELD** statement in an ESCORT script.

The following example indicates the contents of an output file created using the *fldgen* script from two synchronous host application screens. The first synchronous host application screen contains two fields and the second screen three fields; the generated field variables for each screen are separated by a blank line in the output file.

```
FIELD (5,10,12)fld0001     /* UNPR ALPH NORM DISP TAGR (330)
FIELD (10,10,5)fld0002     /* UNPR NUMR NORM DISP TAGR (730)

FIELD (3,10,8)fld0001      /* UNPR ALPH HILT DISP TAGR (170)
FIELD (5,10,10)fld0002     /* UNPR ALPH NORM DISP TAGR (330)
FIELD (15,8,3)fld0003      /* UNPR NUMR DARK TAGR (1128)
```

## Program Listing

```
/***************************************************************************/
/*
/*  This program generates field statements from the current
/*  screen.  To generate a file containing field variables,
/*  you have to be on that screen. From the UNIX prompt, enter:
/*
/*             ESCORT              FLDGEN         outfile[,ALL]
/*
/*  where 'outfile' is the name of the file that will contain the
/*  field statements for the current screen.  The optional parameter
/*  ALL can be used to generate statements for all fields,
/*  i.e., protected and unprotected.  If you omit this parameter,
/*  then statements for unprotected fields only are generated.
/*  This script has been changed for the 3B version of ESCORT.
/*  The script is in a loop where the user is allowed to position
/*  themselves on the screen for field generation, they will
/*  then PRESS ESC f 2 and the script will then generate the field
/*  statements for that screen. The script will then allow the
/*  user to go to other screens and repeat the process.
/*  All field statements generated will be placed in the file
/*  specified by the user at the time of execution. Each format will
/*  be separated by a blank line.
/*
/*
/*  BUGS:
/*  1. If two or more consecutive attributes are present, then
/*     the length of the field may be incorrect.
/*  2. If a field is wrapped (from last field to first field),
/*     then two field statements are generated, i.e., last field
/*     and first field, instead of one contiguous field.
/*
/***************************************************************************/

fldgen          PROG            main

main            SCRIPT
                INT             i              /* field position
                INT             j              /* number of fields
                INT             k
                INT             FIRSTFID       /* 1st attr found flag
                CHAR            (5) P          /* protected/unprotected
                CHAR            (5) A          /* alphanumeric/numeric
                CHAR            (5) H          /* highlighted/normal
                CHAR            (5) D          /* displayable/dark
                CHAR            (5) M          /* data tag set/reset
                CHAR            (4) s
                CHAR            (2) r
                CHAR            (2) c
                CHAR            (4) l
                CHAR            (8) f
                CHAR            (80) line      /* line buffer
                CHAR            (40) file      /* filename
                CHAR            (10) opt       /* ALL option
                INT             row            /* row
                INT             col            /* column
                INT             len            /* length
                INT             olen           /* old length
                INT             all            /* print all fields flag
                INT             total          /* total length
```

```
/***************************************************************************/
/*
/*              check input parameters and open 'outfile'
/*
/***************************************************************************/
                FILE = "&&1"         /* outfile name
                opt = "&&2"         /* all option
                IF file = ""        /* no outfile name specified
                THEN
                 WINDOW          (21,20,23,62)
                 WTO   "    USAGE: ESCORT FLDGEN outfile(,ALL)"
                 EXIT(tutorial)
                 ABEND
                ENDIF

                OPEN (f1,"&&1",a) /* open file to append,handles multiple screens
                IF SYSRET != 0
                THEN                              /* outfile open failed
                WINDOW (21,20,23,62)
                WTO   "      cannot open output file - &&1"
                EXIT(tutorial)
                ABEND
                ENDIF

                SWITCH (opt)                          /* check ALL option
                CASE ""                          /* null option
                 all = 0                              /* unprotected fields only
                CASE "all"                       /* all fields
                 all = 1
                CASE "ALL"                       /* all fields
                 all = 1
                DEFAULT
                 WINDOW          (21,20,23,62)
                 WTO   "    USAGE: ESCORT FLDGEN outfile(,ALL)"
                 EXIT(tutorial)
                 ABEND
                ENDC

/***************************************************************************/
/*
/*  $NEXTFLD returns 0 if no field is found.
/*  $NEXTFLD returns 1921 if there is an attr. byte at position 1920.
/*  case 0" means last field wrapped (no attr at position 1).
/*  case 1" is normal formatted screen field.
/*
/***************************************************************************/

   while(1)
   do
                WINDOW (19,15,23,70)
                WTO   "  Select Application Screen and - "
                WTO   "  Press ESC f 2 to generate field variables,  OR"
                WTO   "  Log Off and exit ESCORT (ESC f 1) to quit."
                exit

                i = 1                       /* initial position
                j = 0                       /* fields counter
                k = 0                       /* Next field position
                FIRSTFID = 1                /* New Screen - NO ATTR Found Yet

                WHILE ((i != 0) & (i != 1921))
                DO
                 i = $NEXTFLD(i)
                 IF                  (i != 0) & (i != 1921)
```

```
THEN
 IF  (FIRSTFID) & (i != 2) & (k = 0)
 THEN i = 1                        /* no attr at position 1
    WTO  "case 0"
 ELSE
      j = (j+1)                    /* bump fields count
      WTO  "case 1"       /* attr at position 1
 ENDIF
 FIRSTFID = 0

 IF  $ATTR(i,P) THEN P = " PROT" ELSE P = " UNPR" ENDIF
 IF  (P = " PROT") & (all = 0)  /* Unprotected fid only
 THEN
     j = (j-1)                     /* reset field count
     cycle
 ENDIF

 IF  $ATTR(i,A) THEN A = " ALPH" ELSE A = " NUMR" ENDIF
 IF  $ATTR(i,H) THEN H = " HILT" ELSE H = " NORM" ENDIF
 IF  $ATTR(i,D) THEN D = " DARK" ELSE D = " DISP" ENDIF
 IF  $ATTR(i,M) THEN M = " TAGS" ELSE M = " TAGR" ENDIF

 CALL  rowcol(i, row, col)

 s   = $ITOS(row)
 IF  $LENGTH(s) = 1 THEN r = ("0" + s) ELSE r = s ENDIF

 s   = $ITOS(col)
 IF  $LENGTH(s) = 1 THEN c = ("0" + s) ELSE c = s ENDIF

 k   = $NEXTFLD(i)
 len = (k - i - 1)
 IF  k = 0  THEN  len = (1922-i-1)  ENDIF /* EOF
 IF  k = 1921 THEN len = (1921-i-1) ENDIF /* NORM
 total = (total + len)
 s   = $ITOS(len)

 k   = $LENGTH(s)/* fill '0s' in length field
 SWITCH (k)
 CASE          1
  l  = ("000" + s)
 CASE          2
  l  = ("00" + s)
 CASE          3
  l  = ("0" + s)
 DEFAULT
  l  = ("" + s)
 ENDC

 if ( j < 1)
 then
  j = 1
 endif
 s   =          $ITOS(j)

 k   =          $LENGTH(s)     /* fill '0s' in field names
 SWITCH (k)
 CASE          1
  f  = ("fld000" + $ITOS(j))
 CASE          2
  f  = ("fld00" + $ITOS(j))
 CASE          3
  f  = ("fld0" + $ITOS(j))
 DEFAULT
```

```
                 f  = ("fld" + $ITOS(j))
                ENDC

                line = ("        FIELD   (" +
                      r + "," + c + "," + l +
                      ") " + f + "     /*" +
                     P + A + H + D + M +
                     " (" + $ITOS(i) + ")")
                IF  (P = " UNPR") | (all = 1)
                THEN
                    WRITE (fl, line)              /* write a line
                ENDIF
               ENDIF
              ENDO

/*************************************************************************/
/*
/*  case of an unformatted screen (no attributes)
/*
/*************************************************************************/
              IF (i = 0) & (j = 0)            /* un-formatted screen
              THEN
               total = (total + 1920)
               line =           ("        FIELD   (01,01,1920) fld0001" +
                                 "      /* UNPR ALPH NORM DISP TAGR (1)")
               WRITE (fl, line)  /* write a line
               line =           ("                                    " +
                                 "      /* UN-FORMATTED SCREEN")
               WRITE (fl, line)  /* write a line
              ENDIF

/*************************************************************************/
/*
/*   case of an unformatted screen (no attributes)
/*
/*************************************************************************/
              IF (i = 1921) & (j = 0)         /* only attr at 1920
              THEN
/*            WTO "case 2"
               total = (total + 1919)
               line =           ("        FIELD   (01,01,1919) fld0001" +
                                 "      /* UNPR ALPH NORM DISP TAGR (1)")
               WRITE (fl, line)  /* write a line
              ENDIF

/*************************************************************************/
/*
/*  print attributes at position 1920
/*
/*************************************************************************/
              IF (i = 1921)      /* attr at position 1920
              THEN
/*            WTO "case 3"
               IF  $ATTR(i,P) THEN P = " PROT" ELSE P = " UNPR" ENDIF
               IF  $ATTR(i,A) THEN A = " ALPH" ELSE A = " NUMR" ENDIF
               IF  $ATTR(i,H) THEN H = " HILT" ELSE H = " NORM" ENDIF
               IF  $ATTR(i,D) THEN D = " DARK" ELSE D = " DISP" ENDIF
               IF  $ATTR(i,M) THEN M = " TAGS" ELSE M = " TAGR" ENDIF
               j    = (j+1)      /* bump attr count
               line =           ("                                    " +
                                 "       /*" + P + A + H + D + M +
                                               " (" + $ITOS(1920) + ")")
               IF  (P = " UNPR") | (all = 1)
               THEN
```

```
                                   WRITE (fl, line)/* write a line
                ENDIF
               ENDIF

               line = (" ")           /* blank line at the end
               WRITE (fl, line)   /* write a line

   endo        /* MAIN WHILE LOOP

/***************************************************************************/
/*
/*   diagnostics
/*
/***************************************************************************/
/*             total = (total + j)              /* add number of attr
/*             line  = ("TOTAL LENGTH = " + $itos(total))
/*             WRITE (fl, line)   /* write total length
/*             line  = ("NUM OF ATTR  = " + $itos(j))
/*             WRITE (fl, line)   /* write value of attr

               ENDS

/***************************************************************************/
/*
/*             This routine returns row and column position
/*
/***************************************************************************/
rowcol         SCRIPT (int cur, int row, int col)/* calc. row & col
               INT offset
               offset= (cur-1)
               row   = ((offset/80)+1)
               col   = (offset-(80*(row-1))+1)
               ENDS

/***************************************************************************/
               copy "/usr/escort/slib/aid_cc"

               ENDP
```

# Get Fields

The *getflds.s* script facilitates the parsing of input records into fields. It is used to read variable length records, that may contain variable length fields. The fields within the record must be delimited by a vertical bar ( | ) field separation character.

To include the *getflds.s* subroutine in your script, add the following **COPY** command:

```
COPY "/usr/escort/common/getflds.s"
```

You must perform a **READ** operation before calling the *getflds.s* subroutine.

The subroutine is invoked in your script by using a **CALL** command.

The *getflds.s* subroutine parses variable length input strings and populates a field table named *fld_tbl*. The values assigned to the *fld_tbl* array can then either be addressed directly or can be assigned to suitable variables within your script.

The following declarations are required in the global variable declarations section of the calling program:

```
CHAR (15)  fld_tbl (20)
```

The *fld_tbl* array is defined with a maximum field length of 15 characters and with a maximum of 20 fields per record. The field length and the number of table entries can be amended to suit your needs. When values are assigned to the array, *fld_tbl (1)* will contain the first field and *fld_tbl (n)* will contain the *n*th field.

```
CHAR (80)  inbuf
```

The *inbuf* variable is the input buffer into which the records are read. The maximum record size contained in the file is 80 characters. Similarly, the record size can be amended to suit your needs.

The variable names may also be amended to suit your particular application.

# Program Listing

```
/************************************************************************
/*                              GETFLDS
/*  This function will parse an input string, delimited by "|"s
/*  individual fields.  These fields will be stored in an array called
/*  "fldtbl", each element containing the field value as it was
/*  encountered.
/************************************************************************
getflds script

                int i
                int indx
                int e                            /* end of field position
                int b                            /* beginning of fld
                int l                            /* length of field
                int len                          /* length of string
                char (80)  string

                indx = 0
                len = $length(inbuf)     /* contents of record read

                while (len != 0)
                do
                e = 0
                indx = (indx+1)
                for i=1 to len
                do
                                  if ($gsubstr(inbuf,i,1) = "|")
                                  then
                                                 e = (i-1)
                                                 break
                                  endif
                endo

                if (e = 0)        /* "|" not found, last field
                then
                                  e = i
                                  fld_tbl(indx) = $gsubstr(inbuf,1,e)
                                  len = 0
                else
                                  fld_tbl(indx) = $gsubstr(inbuf,1,e)
                                  b = (e+2)
                                  if (b > len)
                                  then
                                                 len = 0
                                                 goto ENDLOOP
                                  endif
                                  l = (len-(e+1))
                                  inbuf = $gsubstr(inbuf,b,l)
                                  len = $length(inbuf)
                endif

 ENDLOOP: string = fld_tbl(indx)
                  fld_tbl(indx) = $strip(string)

                  endo

ends
```

# Asynchronous Host Soft Function Keys

In the asynchronous host environment, many applications use the soft function keys, F1 to F8. ESCORT allows the use of these keys from within a script by using **PF1** to **PF8**, or AID keys 1 to 8, to transmit the soft function keys to the asynchronous host.

The escape sequences sent by ESCORT when these commands or keys are used, are the defaults specified for a VT100 terminal. In some instances, applications will define alternate key sequences for the soft functions keys. In this case, use of the **PF1** to **PF8**, or AID keys 1 to 8, will not provide the correct function.

In order to transmit the F1 to F8 keys in this situation, you must use the **$HEX** function to send the appropriate escape sequences to the asynchronous host.

The *fkeys.p* sample program provides an illustration of how the **$HEX** function is used to send the soft function keys to the host.

The sample script includes dummy **SERINIT** parameters which must be amended for your particular asynchronous session. A sample **CALL** command, which sends soft function key F2 default values to the asynchronous host, is also shown.

# Program Listing

```
/*****************************************************************
/* PURPOSE: This sample program demonstrates the use of the
/*  $HEX function to send any string of data to the
/*  host.
/*
/*  The use of the $HEX function is very effective
/*  when you need to send the soft function keys
/*  (F1-F8) to the host and the host has modified
/*  the default VT100 values for those keys.
/*  In that case, you need to be able to send the
/*  sequence of characters that the host application
/*  expects to receive for the keys F1-F8.
/*****************************************************************

fkeys prog main(A1)

                char(3)             F1
                char(3)             F2
                char(3)             F3
                char(3)             F4
                char(3)             F5
                char(3)             F6
                char(3)             F7
                char(3)             F8

main script

                F1 = $HEX ("1b4f50")              /* ESC OP */
                F2 = $HEX ("1b4f51")              /* ESC OQ */
                F3 = $HEX ("1b4f52")              /* ESC OR */
                F4 = $HEX ("1b4f53")              /* ESC OS */
                F5 = $HEX ("1b4f54")              /* ESC OT */
                F6 = $HEX ("1b4f55")              /* ESC OU */
                F7 = $HEX ("1b4f56")              /* ESC OV */
                F8 = $HEX ("1b4f57")              /* ESC OW */

                serinit(1,1200,e,1,7,full,"host1","")
                connect(a1)
                if (sysret = -1)
                then
                                  log "Connection to HOST1 failed"
                                  return
                endif

                                  .
                                  .
                                  .
                                  .
                                  .
                                  .
                call sendfkey(2)                  /* Send F2 */
                                  .
                                  .
                                  .
                                  .
                                  .

ends
```

```
/*********************************************************************
/*                SENDFKEY
/*              PURPOSE: Send ASCII Soft Function Keys F1-F8
/*********************************************************************

sendfkey script(int key)

                switch(key)
                case 1
                                text F1
                case 2
                                text F2
                case 3
                                text F3
                case 4
                                text F4
                case 5
                                text F5
                case 6
                                text F6
                case 7
                                text F7
                case 8
                                text F8
                default                               /* Default required */
                endc

ends

endp
```

# 6 Local Screen Generator Utility Program


**Overview** 6-1

**Accessing and Quitting LSGEN** 6-3
Accessing LSGEN 6-3
Operator Information 6-5
On-Line Help 6-6
Quitting LSGEN 6-7

**Creating and Editing Fields** 6-9
Edit Mode 6-9

**Defining Fields** 6-19
Field Definition Mode 6-19

**LSGEN Error Messages** 6-27

**LSGEN Key Sequences** 6-29
Special Key Combinations 6-29
LSGEN Cursor Movement Keys 6-32

# Overview

This chapter contains information you need to know to use the ESCORT Local Screen Generator (LSGEN) Utility Program.

LSGEN is a full screen editor program that allows you to create local screen formats for subsequent use with an ESCORT program. The formatted screens may contain a variety of field attributes. Formatted screens created using LSGEN are saved on your system as standard UNIX files and can be accessed using your system editor utility. Local Screen files can also be retrieved and modified by the LSGEN program and can be included in an ESCORT program by use of the ESCORT **COPY** command.

Local screens formats are created independently of any ESCORT program and, therefore, you do not require a knowledge of ESCORT to be able to create local screen formats using LSGEN.

This chapter is divided into four sections:

- accessing and quitting LSGEN
- modes of operation of the program
- LSGEN error messages
- key sequences specific to LSGEN.

**Note**
LSGEN is a separate utility program and, therefore, it utilizes a separate set of key sequences which may differ from the key sequences used in ESCORT.

A demonstration local screen format file, named ***/usr/escort/common/demoscrn*** is available, as part of the LSGEN utility program, on your system. Access this demonstration screen to test the various features of LSGEN.

When you have read this chapter you will be able to create local screen formats, using all of the field attributes available with your system, for use with an ESCORT program.

# Accessing and Quitting LSGEN

This section provides you with information on accessing LSGEN from the UNIX shell and on quitting the LSGEN program. Information regarding the on-line help screen and the operator information area is also reviewed.

## Accessing LSGEN

The procedure for invoking LSGEN from the UNIX shell is described below.

1 On the command line, type

   ***lsgen file_1 [file_2]***

   and press RETURN.

   *file_1* specifies the name of the local screen format input file. The *file_1* may be either a new file or an existing file. LSGEN checks the format of the file and displays an error message if it does not conform to the syntax rules required by ESCORT.

   *file_2* specifies the name of the local screen format output file. The *file_2* may be either a new file or an existing file. The *file_2* is an optional parameter and, if omitted, *file_1* is used as the output file.

   If the output file exists the contents are overwritten when you quit LSGEN and save the generated output.

   LSGEN automatically assigns the output file name to the *screen_name* operand of the **BEGFMT** statement in the generated local screen format.

2 An LSGEN banner screen is displayed. Press RETURN to continue.

3 If the output file, *file__2*, has been specified on the command line and this file exists, LSGEN displays the following warning message:

**Output file** *file__2* **exists. You may overwrite it.**
**RETURN=continue ESC=quit**

4 LSGEN checks the read and write permissions to the files specified on the command line. If you do not have read permission for *file__1*, LSGEN displays the following error message and quits; control is returned to the UNIX shell:

**Cannot open** *file__1* **file.**

If you do not have write permission for the output file, *file__1*, (or *file__2* if specified) LSGEN displays the following message:

**You do not have write permission to the** *file__n* **file.**

**Updates are not allowed, you may only view the format.**

The *file__n* specifies the output file.

5 If *file__1* is a new file, LSGEN displays a blank, unformatted screen.

6 If *file__1* is an existing file, LSGEN displays a summary of the local screen format with the following information for each field statement contained in the *file__1* file:

- statement line number
- sequential field statement number
- field row, column and length
- field attributes
- format name and field name
- the flag, *STR*, indicating that the field has been initialized with a string.

A warning is also displayed if any field has been initialized with a character string longer than the defined field length.

Press (RETURN) to display the contents of *file__1* as a formatted screen.

## Operator Information

LSGEN displays certain operator information messages in the operator information area. If your terminal has a 24-line screen, the operator information area must first be toggled on. Press (ESC) **I** to toggle on the operator information area. The operator information area is automatically displayed on terminals with 25-line screens. The operator information area can be toggled off or on by pressing (ESC) **I**.

In addition to error messages and operator prompts, the operator information area displays

**ESC 1=HELP row: col:**

The row and column location of the current cursor position are displayed.

## On-Line Help

While in Edit mode, an on-line help screen is available. Edit mode is discussed in the next section. Press ESC **1** to display the on-line help screen. The help screen

- summarizes the special function keys and other key combinations that you can use in LSGEN
- summarizes the cursor movement keys that are available in Edit mode.

Press any key to return to Edit mode from the help screen.

## Quitting LSGEN

You can quit the LSGEN program, in Edit mode, and either save or cancel the generated local screen format.

- To save the generated local screen format and return to the UNIX shell, press (ESC) **2**. The local screen format is written to either *file_1* or to *file_2*, depending on the files specified on the command line. See the section "Accessing LSGEN" for further information.
- To exit LSGEN and return to the UNIX shell without saving the contents of the generated local screen format, press (ESC) **q**. The LSGEN program requests confirmation that the generated local screen format is not to be saved.

# Creating and Editing Fields

The two modes of operation of LSGEN, Edit mode and Field Definition mode, are discussed in this section. Features covered include creating a local screen format; deleting and inserting characters and lines; selecting and copying, moving, and deleting fields; and defining fields.

## Edit Mode

Edit mode provides the functions of a screen editor. It is the default mode when LSGEN displays either a blank screen when creating a new local screen format, or a formatted screen when modifying an existing local screen format.

You can move the cursor to any position on the screen and type uppercase and lowercase text, numbers, and special characters. Type narrative, either before or after creating fields, at the appropriate location to pre-initialize the field.

Various functions are available that provide you with the ability to insert and delete individual characters, lines of text, and fields. These functions are described in this section.

## LSGEN Cursor Movement

Cursor movement is controlled by use of the following key combinations and function keys:

| Key | Function |
|---|---|
| ↑ * or CTRL – t | **Up arrow** Moves the cursor up one line. |
| ↓ * or CTRL – v | **Down arrow** Moves the cursor down one line. |
| ← * or CTRL – f | **Left arrow** Moves the cursor one position to the left. |
| → * or CTRL – g | **Right arrow** Moves the cursor one position to the right. |
| SPACE BAR | **Space bar** Moves the cursor one position to the right and displays a blank space. |
| RETURN | **Return (New line)** Moves the cursor to the first position on the next line. |
| BACK SPACE * | **Back space** Moves the cursor one position to the left and deletes the character. |
| TAB | **Tab** Moves the cursor to the beginning of the next tab position. Tab positions are set at every eighth column. |

* These keys must be defined in the UNIX system, *terminfo*, terminal information files.

## Insert Characters

Entering characters normally *overtypes* any existing characters on the screen. Press ESC **i** to *insert* characters at the current cursor position. LSGEN displays

**INS**

in the operator information area when *insert* is selected. ESC **i** toggles between *insert* and *overtype* modes of typing.

Characters can be inserted inside a field or into a string located in an unformatted area of the screen. When inserting characters, characters to the right of the cursor are shifted. The shift area extends to the end of the current field or to the end of the current line, depending upon whether insertion is inside or outside a field. Characters are lost if they are shifted out of the shift area.

A field cannot be shifted off the current line by inserting characters to the left of, or inside the field.

## Delete Characters

Press ESC **X** to delete individual characters at the current cursor position. Characters can be deleted from within a field or from a string located in an unformatted area of the screen. When deleting characters, characters to the right of the cursor are shifted. The shift area extends to the end of the current field or to the end of the current line, depending upon whether deletion is inside or outside a field.

Characters cannot be deleted from the left of a field that wraps around the current line.

Field attribute bytes and field termination characters, (<), cannot be deleted.

The delete character key sequence, ESC **X**, can be used in insert mode.

## Insert and Delete Lines

To insert a blank line at the current cursor position, press ESC **o**. The screen area following the inserted line scrolls down. You cannot insert blank lines inside a field that wraps around the current line, nor can fields be shifted off the bottom of the screen.

To delete a line at the current cursor position, press ESC **d**. The screen area following the deleted line scrolls up and a blank line is inserted at the bottom of the screen. You cannot delete a line that contains a field.

## Create Fields

Every local screen field is preceded by an attribute byte. The attribute byte occupies a single screen position. To create a new field:

1. Position the cursor at the screen location immediately before the first position of the field to be created. Press (ESC) **4** and the attribute byte is displayed at the current location. LSGEN also displays

   **Field not terminated.**

   in the operator information area. The attribute byte overwrites any character at the current screen location. Refer to Appendix D "Interpretation of Attribute Bytes" for information on interpreting field attribute bytes.

2. Extend the field created using the cursor movement keys. Refer to the row and column indicators displayed in the operator information area to determine the correct length of the field.

3. Terminate the field. Press (ESC) **4** and the field termination character (<) is displayed at the first screen location following the last character in the field.

4. You may now choose to define the name and attributes for the field created. Position the cursor at any location within the field and press (ESC) **3**. LSGEN enters Field Definition mode and allows you to specify the field name, define the attributes and redefine the field length if necessary. Refer to "Defining Fields" in this section for further information.

   Defining the field is optional; if you do not define the field,

   a. LSGEN automatically assigns the current field attributes to the field. Current field attributes are those attributes last specified in Field Definition mode. If current field attributes have not been specified in Field Definition mode, LSGEN assigns the default field attributes. Refer to the attribute tables listed in the **FIELD** statement detailed in Chapter 4 for information on default attributes.

**b** The default field name *dummy* is assigned to the field when the local screen format is saved.

LSGEN checks to ensure that fields are not created with zero length and that they do not overlap.

## Delete Fields

To delete a field, position the cursor at any location within the field and press ESC **6**. LSGEN displays

**Delete the field? RETURN=YES ESC=NO**

in the operator information area. Press RETURN and the attribute byte and the field termination character (<) are deleted. Any literal characters previously contained within the field are not deleted, however, allowing you to move the literal string and recreate the field in another location. Literal string characters may be overtyped or deleted. Refer to the section "Delete Characters" in this chapter.

Press ESC if you do not want to delete the field.

You can also delete a field that has not been terminated by pressing ESC **6**.

## Copy Fields

To copy an existing field:

1. Position the cursor at any location within the field to be copied and press (ESC) **5**. LSGEN copies (yanks) the field length, attributes and any literal character string contained within the field.
2. Position the cursor at the screen location of the attribute byte for the new field and press (ESC) **5**. LSGEN displays (puts) a copy of the field in the new location. The field length, attributes and literal character string, if any, duplicate the original field. LSGEN assigns the default field name *dummy* to the new field. The field name can be amended using the Field Definition mode.

LSGEN checks to ensure that sufficient unformatted screen space exists to accommodate the copied field.

## Move Fields

To move an existing field to a new location, use a combination of the LSGEN Copy Field, (ESC) **5**, Delete Field, (ESC) **6**, and Define Field, (ESC) **3**, key combinations.

1. Select the field to be moved and copy (yank) the field using (ESC) **5**.
2. Put a copy of the field in the new location using (ESC) **5**.
3. Delete the original field using (ESC) **6**.
4. Redefine the field name of the new field using (ESC) **3**.

## Field Display

LSGEN provides four functions that assist in visualizing formatted screens:

- Press ESC **7** to toggle on the attribute bytes and field termination characters (<) for all fields. Use ESC **7** to toggle on and off the attribute bytes and field termination characters. Refer to Appendix D "Interpretation of Attribute Bytes" for information on interpreting field attribute bytes.
- Press ESC **8** to display all fields using the defined visual attributes: Intensity, (Primary Attribute, Group 3), and Display, (Extended Field Attribute, Group 5). Use ESC **8** to toggle on and off the display of visual attributes.

  LSGEN displays

  **FMT**

  in the operator information area when display attributes mode is selected.
- Press ESC **f** to display all unprotected fields, that do not contain a literal character string, with dot ( . ) fill characters. The fill character is not part of the formatted screen and is not saved to the file. Use ESC **f** to toggle on and off display of the fill character.

  LSGEN displays

  **FIL**

  in the operator information area when fill mode is selected.
- Press ESC **r** to refresh the entire screen. Use this feature to repaint the screen to remove, for instance, unwanted UNIX system messages that may appear at the terminal when you are creating a formatted screen.

LSGEN provides the option of clearing the entire screen. *Note that all screen data are lost and are not saved to the file.* To clear the screen, press ESC **z**.

# Defining Fields

## Field Definition Mode

Field Definition mode is entered from Edit mode. Field Definition mode allows you to define or modify a field and its attributes. This section contains information on defining all the characteristics of a field.

## Enter Field Definition Mode

The following procedure shows you how to enter Field Definition mode:

- ☐ Every local screen field is preceded by an attribute byte. To create a new field, position the cursor at the screen location immediately before the first character in the field. The attribute byte occupies this position.
- ☐ To define the attributes of a field created by use of the Create Field key combination, (ESC) **4**, position the cursor anywhere within the field.
- ☐ To modify an existing field, position the cursor anywhere within the field.

When the cursor is positioned correctly, press (ESC) **3**. The following menu is displayed in a window:

```
Field name:
    row:        col:
    length:

Protected
Data type
Intensity
Data tag
Display
Foreground
Background

TAB=skip field             ESC=cancel
SPACE BAR=toggle           RETURN=save
```

The field name, length and attributes are displayed if you are modifying an existing field.

For new fields, the field name is blank, the length is shown and current field attributes are displayed. Current field attributes are those attributes last specified in Field Definition mode. If current field attributes have not been specified in Field Definition mode, LSGEN assigns the default field attributes.

The following sections describe the procedure for defining or modifying a field.

## Define Name

Enter the field name in the following format:

**[format.]field_name**

where **format** specifies an optional screen format name to identify uniquely a field name that may appear in multiple formats.

**field_name** specifies the simple field name. The *format* and *field_name* may each be up to eight alphanumeric characters, and the first character must be alphabetic. The *format* is optional, but when it is included it must be separated from the *field_name* by a dot ( . ).

If the field name is not specified, LSGEN automatically assigns the default name *dummy* to the field when the local screen format is saved. This default value corresponds to the keyword **DUMMY** used in the ESCORT **FIELD** statement, and is used to declare a literal field.

You can modify an existing field name by overtyping and using (BACKSPACE) if necessary.

Press (TAB) or (↓) to skip to the *length* field. LSGEN checks the field name entered and displays

**Duplicate name.**

in the operator information area if the field name entered is not unique. Press (RETURN) and amend the field name.

## Define Length

Enter the new field length, or overtype the existing field length. Valid field lengths are in the range from 1 to 1919. LSGEN displays

**Overlapping fields.**

in the operator information area if the length entered would cause the current field to overlap an existing field.

Press (TAB) or (↓) to skip to the attributes fields. Press (↑) to return to the previous field.

The field length may be left as *undefined*. However, if the field length is not defined, when you exit from Field Definition mode, LSGEN displays

**Field not terminated.**

in the operator information area. You must either

- extend the field, using the cursor movement keys, and terminate the field, by pressing (ESC) **4**, or,
- delete the field by pressing (ESC) **6**.

See "Create Fields" and "Delete Fields" in the "Edit Mode" section of this chapter for further information.

## Define Attributes

Press the Space Bar, (←) or (→) to cycle through all of the available attribute options.

Press (TAB) or (↓) to skip to the next attribute field. Press (↑) to return to the previous field.

You can select one attribute, from the following tables, for each of the seven groups. Note that *Code* relates to the attribute code that LSGEN writes to the field statement in the local screen format file.

| **Primary Attribute - Group 1** | |
|---|---|
| Protected | Code |
| Yes | P |
| No | U |

| **Primary Attribute - Group 2** | |
|---|---|
| Data type | Code |
| Numeric | N |
| Alphabetic | A |

| **Primary Attribute - Group 3** | |
|---|---|
| Intensity | Code |
| Normal | N |
| Highlighted | H |
| Dark | D |

| **Primary Attribute - Group 4** | |
|---|---|
| Data tag | Code |
| Modified | M |
| Reset | R |

| Extended Field Attribute - Group 5 | |
|---|---|
| **Display** | **Code** |
| Normal | N |
| Blink | B |
| Reverse video | R |
| Underline | U |

| Extended Field Attribute Foreground - Group 6 | |
|---|---|
| **Color** | **Code** |
| Black | 0 |
| Blue | 1 |
| Green | 2 |
| Cyan | 3 |
| Red | 4 |
| Magenta | 5 |
| Brown | 6 |
| White | 7 |
| Gray | 8 |
| Light blue | 9 |
| Light green | 10 |
| Light cyan | 11 |
| Light red | 12 |
| Light magenta | 13 |
| Yellow | 14 |
| Hi-lit white | 15 |

| Extended Field Attribute Background - Group 7 | |
|---|---|
| Color | Code |
| Black | 0 |
| Blue | 1 |
| Green | 2 |
| Cyan | 3 |
| Red | 4 |
| Magenta | 5 |
| Brown | 6 |
| White | 7 |

**Note**
The foreground and background colors, defined by the Extended Field Attribute, Groups 6 and 7, are not available in the UNIX operating system version of LSGEN. They are defined for local screen format compatibility between the UNIX operating system version and the MS-DOS operating system version of LSGEN.

Note that if you select *Dark* Intensity from the Primary Attribute, Group 3, the foreground color, defined in Extended Field Attribute, Group 6, automatically changes to the background color defined in Extended Field Attribute, Group 7.

## Exit Field Definition Mode

Amend any fields as necessary by using TAB, ↑ or ↓ to move between fields.

Choose one of the following options to exit Field Definition mode and return to Edit mode:

- If you are satisfied that the field is correctly defined, press RETURN to save the field and its associated name, length and attributes. The Field Definition window is erased and the field is displayed on the screen at the appropriate location.
- To cancel the generated field, press ESC. The Field Definition window is erased and the defined field is *not* saved.

# LSGEN Error Messages

This section lists error messages that LSGEN writes to the operator information area. A brief explanation of each error message is included.

| Message | Description |
|---|---|
| **Overlapping fields. RETURN=continue** | A field has been defined with a length that will cause fields to overlap. Fields may not overlap, redefine the length or reposition the field. |
| **Cannot delete attribute byte. RETURN=continue** | The delete character key sequence, ESC **X**, was pressed with the cursor positioned at an attribute byte. Attribute bytes cannot be deleted, reposition the cursor. |
| **Cannot shift left. RETURN=continue** | Characters cannot be deleted from the left of a field that wraps around a line. |
| **Cannot scroll. RETURN=continue** | Blank lines cannot be inserted inside a field that wraps around a line. Blank lines cannot be inserted that would cause a field to shift off the bottom of the screen. Lines that contain a field cannot be deleted. |
| **Duplicate name. RETURN=continue** | The field name entered in Field Definition mode already exists. Duplicate field names are not allowed, enter a new field name. |

| Message | Description |
|---|---|
| **Length undefined. RETURN=continue** | Field length has not been defined. Extend the field using the cursor movement keys and terminate the field, or delete the field. |
| **Nothing to delete. RETURN=continue** | The cursor was not positioned within a field when the delete field key sequence, ESC **6**, was pressed. Reposition the cursor. |
| **Cannot shift right. RETURN=continue** | Characters cannot be inserted in a field that would cause the field to shift off the current line. |
| **Name too long. RETURN=continue** | The field name exceeds the maximum allowed length. The *format* and *field name* may each be up to eight characters in length, separated by a dot ( . ). |
| **Bad name. RETURN=continue** | An illegal character (e.g., unprintable character) has been used in a field name. The first character in a field name must be alphabetic. |

# LSGEN Key Sequences

This section lists the special key combinations together with the cursor movement key sequences available in Edit mode in the LSGEN program.

You can program the special function keys, F1 to F8, on your terminal, if available, to simulate the key combinations, ESC **1** to ESC **8**.

## Special Key Combinations

ESC **1** — **On-Line Help**

Displays the on-line help screen. Press any key to return to Edit mode from the help screen.

ESC **2** — **Save screen and return to UNIX shell**

Saves the local screen format generated, quits LSGEN and returns to the UNIX shell.

ESC **3** — **Field Definition mode**

Enters Field Definition mode from Edit mode. You may use this function key to define a new field or to modify an existing field.

**ESC 4** — **Create Field**

Establishes the start of a new field and displays the field attribute byte. When the field has been extended, by use of the cursor movement keys, press ESC **4** to terminate the field and display the field termination character, (<).

**ESC 5** — **Copy Field**

Copies (yanks) an existing field. The field length, attributes and any literal string are copied. Position the cursor at the new screen location, by use of the cursor movement keys, and press ESC **5** to display (put) the copied field. LSGEN assigns the default name *dummy* to the copied field.

**ESC 6** — **Delete Field**

Deletes a field. Literal characters previously contained within the deleted field are not deleted.

**ESC 7** — **Toggle field limit characters**

Toggles on and off field attribute bytes and field termination characters, (<).

**ESC 8** — **Toggle field visual attributes**

Toggles on and off the defined visual attributes, Intensity and Display. **FMT** is displayed in the operator information area when display attributes mode is selected.

**ESC d** — **Delete line**

Deletes a line at the current cursor position. Screen lines that contain fields cannot be deleted. The screen automatically scrolls.

ESC f

**Toggle fill character**

Toggles on and off display of the fill character, ( . ), in all unprotected fields that do not contain literal character strings. **FIL** is displayed in the operator information area when fill mode is selected.

ESC i

**Toggle insert/overtype modes**

Toggles on and off between insert and overtype modes. **INS** is displayed in the operator information area when insert mode is selected.

ESC l

**Toggle operator information area**

Toggles on and off the operator information area, or status line.

ESC o

**Insert blank line**

Inserts a blank line at the current cursor position. The screen automatically scrolls.

ESC q

**Exit LSGEN and return to UNIX shell**

The local screen format is *not* saved, LSGEN quits and control returns to the UNIX shell.

ESC r

**Refresh screen**

Repaints the entire screen with defined fields and character strings.

ESC x
or
CTRL – d

**Delete character**

Deletes individual characters. Field attribute bytes and field termination characters, (<), may not be deleted.

ESC z

**Clear Screen**

Clears the screen; all fields and character strings are deleted.

# LSGEN Cursor Movement Keys

| Key | Description |
|---|---|
| ↑ * or CTRL – t | **Up arrow**<br>Moves the cursor up one line. |
| ↓ * or CTRL – v | **Down arrow**<br>Moves the cursor down one line. |
| ← * or CTRL – f | **Left arrow**<br>Moves the cursor one position to the left. |
| → * or CTRL – g | **Right arrow**<br>Moves the cursor one position to the right. |
| SPACE BAR | **Space bar**<br>Moves the cursor one position to the right and displays a blank space. |
| RETURN | **Return (New line)**<br>Moves the cursor to the first position on the next line. |
| BACK SPACE * | **Back space**<br>Moves the cursor one position to the left and deletes the character. |
| TAB | **Tab**<br>Moves the cursor to the beginning of the next tab position. Tab positions are set at every eighth column. |

---

* These keys must be defined in the UNIX system, *terminfo*, terminal information files.

# Appendices


**A** **Error Messages** A-1

**B** **Debugging Facilities** B-1
TRACE Command B-1
DUMP Command B-3

**C** **AID Subroutines Library** C-1

**D** **Interpretation of Attribute Bytes** D-1

**E** **Key Sequences** E-1
Key Sequences for Standard ASCII Terminals E-2
Key Sequences for AT&T 4410 and Teletype 5410 Terminals E-4
Key Sequences for AT&T 4418 and Teletype 5418 Terminals E-6
Key Sequences for AT&T 4425 and Teletype 5425 Terminals E-8
Key Sequences for AT&T 605 Business Communications Terminal E-10
Key Sequences for AT&T 610, 615, 620, and 630 Business Communications Terminals E-12
Key Sequences for DEC VT100 Terminal E-14

**F** **Environment Variables and Customization** F-1
Setting 3270 Emulator+ ESCORT Environment Variables F-2
Terminal Customization F-4

---

**G** **Additional Programs** G-1
Writing a Tutorial Script G-2
Performing Regression Testing G-4
Reading from a Pipe File G-7
Writing to a Pipe File G-8

# Error Messages

This section lists all numbered ESCORT error messages in numerical order, together with a partial list of the more common run time error messages. A brief explanation of each error message is included.

## ESCORT Error Messages

If an error occurs during syntax checking, ESCORT prints the line number, external file name, script name, and source (i.e., the operand/operator) of the error to a file, in the directory defined by the ESCDIR environment variable, named *escort.pr{proc-id}*, where *{proc-id}* refers to the unique process identification the UNIX operating system assigns to each session.

If an error occurs during execution, ESCORT prints the name of the script in which the error was detected, the name of the script which called that script, and the command causing the error, also to the *escort.pr{proc-id}* file.

Following is a list of all numbered error messages. All ESCORT error messages are preceded by the literal *ECS*; for example, ECS000.

| No. | Error Message | Description |
|---|---|---|
| 000 | **dummy msg** | Error in ESCORT. Call the AT&T Hotline. |
| 001 | **variable not defined** | The variable used has not been previously defined by a **CHAR, INT**, or **FIELD** statement. |
| 002 | **row invalid** | The value of a row variable is not between 1 and 24. |
| 003 | **column invalid** | The value of column variable is not between 1 and 80. |
| 004 | **invalid command** | A valid command is expected but not found. |
| 005 | **invalid operator** | A valid arithmetic or logical operator is expected but not found. |
| 006 | **string required** | A string type operand is required but not found. |
| 007 | **expected '('** | An open parenthesis is required. |
| 008 | **expected numeric** | A numeric value is required. |
| 009 | **expected ')'** | A close parenthesis is required. |
| 010 | **invalid identifier** | An identifier must start with an alphabetic character and contain no more than 8 characters. The exception to this rule is a field variable. |
| 011 | **operator required** | An arithmetic (e.g., +) or logical operator (e.g., >) is required. |
| 012 | **type conflict** | String used in an integer context or an integer in a string context. |
| 013 | **THEN required** | Keyword **THEN** is missing in an **IF** statement. |
| 014 | **label after GOTO** | Invalid label follows a **GOTO** command. |

| No. | Error Message | Description |
|---|---|---|
| 015 | **type not implemented** | Error in ESCORT. Call the AT&T Hotline. |
| 016 | **operator invalid** | An arithmetic (e.g., +) or logical (e.g., >) operator is required. |
| 017 | **invalid character** | Program contains an illegal special character. |
| 018 | **IF not terminated** | At least one **IF** statement in your script is missing an **ENDIF**. |
| 019 | **invalid syntax** | A syntax error occurred, but ESCORT cannot give you the precise definition of the error. |
| 020 | **name > 8 characters** | A name is too long. |
| 021 | **invalid subscript** | An array subscript must have a numeric value of less than or equal to 2048. |
| 022 | **script name required** | After an **ENDS** (end of script) statement, a script name is expected unless an **ENDP** (end of program) is present. |
| 023 | **expected command** | A valid command is expected. |
| 024 | **operand > 132 characters** | A literal may not exceed 132 characters. |
| 025 | *****next addr invalid** | Error in ESCORT. Call the AT&T Hotline. |
| 026 | *****command code null** | Error in ESCORT. Call the AT&T Hotline. |
| 027 | **DO required** | Keyword **DO** is missing in a **FOR** or **WHILE** statement. |
| 028 | **DO/ENDO not paired** | Keyword **DO** or **ENDO** is missing within a named script. |
| 029 | **label unknown** | Branch label in a **GOTO** statement is not defined. |

| No. | Error Message | Description |
|---|---|---|
| 030 | **script unknown** | Script name used in a **CALL** statement does not exist in the program. |
| 031 | **expected TO** | Keyword **TO** is missing in a **FOR** statement. |
| 032 | **clause type invalid** | String clause is required instead of an integer clause, or vice versa. |
| 033 | **embedded copy** | Only two levels of embedded **COPY** statements are allowed. (If file A contains a **COPY** for file B and file B contains a **COPY** for file C, file C may not have a **COPY** statement.) |
| 034 | **open failed - script** | The **OPEN** command did not work for the script named on the UNIX command line. This usually happens if there is no file with the specified name. |
| 035 | **storage limit exceeded** | Storage requirements of the program exceeded the dynamic storage area allocated by ESCORT. If this happens, split the program into two programs, if possible, and run them sequentially by using a UNIX shell script. An alternative is to reduce the size and number of variables and constants in the program. |
| 036 | **USER abend** | A user abend was issued. No ESCORT dump is produced. |
| 037 | **identifier already used** | The name has already been used. All global variables must be unique within a program. All local variables within a script must be unique. |

| No. | Error Message | Description |
|---|---|---|
| 038 | **illegal in global sect.** | Only declaration statements and copy statements may be in the global section of the program. |
| 039 | **unallocated variable** | Error in ESCORT. Call the AT&T Hotline. |
| 040 | **parm list mismatch** | Parameters specified in the CALL statement must correspond to parameters defined in the SCRIPT statement in type and number. See the CALL and SCRIPT commands for further detail. |
| 041 | **file name > 40** | External file name (including complete path specification) may not exceed 40 characters. |
| 042 | **prev script not ended** | Script lacks ENDS statement. Each script must terminate with an ENDS statement before another begins. |
| 043 | **PROG required first** | Program lacks PROG statement. Each program must begin with a PROG statement. |
| 044 | **2nd PROG** | Program contains more than one PROG statement. Each program may contain only one PROG statement. |
| 045 | **missing ENDP** | Program lacks ENDP statement. Each program must terminate with an ENDP statement. |
| 046 | **2nd ENDS for script** | Script contains more than one ENDS statement. Each script may contain only one ENDS statement. |
| 047 | **string length > 2K** | A character string may not exceed 2048 bytes. |

| No. | Error Message | Description |
|---|---|---|
| 048 | **2 successive operators** | An expression may not contain two operators in a row without an operand between them (e.g., +/ is an error). |
| 049 | **syntax error** | A syntax error occurred, but ESCORT cannot give you the precise definition of the error. |
| 050 | **missing left paren** | An open parentheses is required. |
| 051 | **2 successive operands** | An expression may not contain two operands in a row without an intervening operator between them. |
| 052 | **time format error** | The correct format is *hh:mm:ss* (hours, minutes, seconds). |
| 053 | **date format error** | The correct format is *mmddyy* or *mm-dd-yy* (month, day, year). |
| 054 | **time/date format error** | See above two error messages. |
| 055 | **bad argument for $LENGTH** | The argument must be either a string or a field variable. |
| 056 | **value > max** | The parameter value specified exceeds the maximum allowable value. |
| 057 | **array initialize error** | An error occurred while processing an array initialization statement. Errors are usually caused when an initial string is assigned to a shorter character element, or when an initial value list has more items than the number of elements in the array. |

| No. | Error Message | Description |
|---|---|---|
| 058 | **table overflow** | Program exceeds capacity of 2500 variables and constants. You may reduce this number by deleting unused field variables from a copied format file. If you cannot reduce the variables or constants, split the program into two programs and run them sequentially in a batch stream. |
| 059 | **$GSUBSTR past string end** | A substring begins after the end of string argument. |
| 060 | | Not used. |
| 061 | **> 40 format names defined** | Only 40 different format names specified by the **FORMAT** command are allowed in a program. |
| 062 | **qualified name invalid** | Program contains invalid name. |
| 063 | **max fields exceeded** | Only 500 fields may be defined on an application screen. |
| 064 | **parse list overflow** | Program exceeds allowable program size. Split the code into two programs and run them sequentially in a batch file. |
| 065 | **no loop to break** | **BREAK** or **CYCLE** statement used improperly. **BREAK** or **CYCLE** can be used only within a *DO/ENDO* loop. |
| 066 | **invalid offset** | Specified position or offset is not within the given string. |
| 067 | **open files > max** | Program exceeds allowable number of open files. You may have 10 files open at the same time. |

| No. | Error Message | Description |
|---|---|---|
| 068 | **invalid file operation** | READ or WRITE command used improperly. Check for one of the following common errors: READ/WRITE before OPEN command, READ issued against a file opened for WRITE, or WRITE issued against a file opened for READ. |
| 069 | **CASE without SWITCH** | CASE statement was found without a prior SWITCH statement. Check proper syntax. |
| 070 | **DEFAULT must be last** | DEFAULT must follow all CASE statements. DEFAULT is a special type of CASE statement within a SWITCH statement. |
| 071 | **missing ENDC for SWITCH** | ENDC (end case) statement must terminate all SWITCH statements. |
| 072 | **terminal input inhibited** | Script calls for entering data in a protected field. This commonly occurs when a script is attempting to enter data on the wrong screen or in a protected field on the correct screen. |
| 073 | **integer overflow** | Integer exceeds allowable maximum value of $-2^{31}+1$ to $+2^{31}-1$. |
| 074 | **parm not valid** | Parameter option, or operand, is improperly specified. |
| 075 | **invalid option** | Option is specified incorrectly. |

| No. | Error Message | Description |
|---|---|---|
| 076 | **invalid window** | Row and column positions are specified incorrectly. Three characters are required for minimum window width and window height. |
| 077 | **too many CASES** | Program exceeds limit of 50 CASES in a SWITCH statement. |
| 078 | **missing THEN** | Required THEN is missing in an IF THEN ELSE statement. |
| 079 | **missing DO** | Required DO is missing in a FOR or WHILE statement. |
| 080 | **CLOSE for closed file** | CLOSE command was issued for an already closed file. |
| 081 | **OPEN for open file** | OPEN command was issued for an already open file. |
| 082 | **READ before open** | READ command was issued for a closed file. |
| 083 | **WRITE before open** | WRITE command was issued for a closed file. |
| 084 | **DO illegal** | DO is allowed only in a FOR or WHILE statement. |
| 085 | **CHKPT before open** | CHKPT command was issued for a closed file. |
| 086 | **CHKPT for read file** | CHKPT command was issued for an input file. |
| 087 | **file 'nickname' required** | File management commands require assignment of an internal name in the OPEN statement. |
| 088 | **file not opened for read** | READ command was issued for a file not opened with the read option. |

| No. | Error Message | Description |
|---|---|---|
| 089 | **file not opened for write** | **WRITE** command was issued for a file not opened with the write option. |
| 090 | **THEN illegal** | **THEN** is allowed only in an **IF** statement. |
| 091 | **no IF for ENDIF** | **ENDIF** statement is not preceded by an **IF** statement. |
| 092 | **zero not allowed** | A zero value is not permitted in this context. |
| 093 | **$ATTR - cursor invalid** | Screen position is incorrect. It must be at the start of a field. |
| 094 | **&& numeric suffix missing** | Numeric suffix is missing. |
| 095 | **invalid mode** | Error in ESCORT. Call the AT&T Hotline. |
| 096 | **illegal in local format** | An invalid statement has been included in a local screen format. A local screen format may contain only **BEGFMT/ENDFMT** and **FIELD** statements. |
| 097 | **invalid code in attr list** | An invalid attribute has been included in a **CHGATTR** or **FIELD** command attribute list operand. Valid attributes are listed in the **FIELD** command in Chapter 4. |
| 098 | **format not built** | Local screen format lacks **ENDFMT** statement. Each local screen format definition area must begin with a **BEGFMT** and end with an **ENDFMT** statement. |

| No. | Error Message | Description |
|---|---|---|
| 099 | **format not found** | Screen name defined by a **GETFMT** not found in *spilled* format files. |
| 100 | **cannot open format spill file** | ESCORT is unable to write the spilled formats to the spill file. You must have write permission for the file. |
| 101 | **cannot open ESCORT Log file escort.lg**{*proc-id*} | ESCORT is unable to open the *escort.lg*{*proc-id*} file specified by a **LOG** command. You must have write permission for the file. |
| 102 | **cannot open ESCORT Capture file escort.cp**{*proc-id*} | ESCORT is unable to open the *escort.cp*{*proc-id*} file specified by a **CAPTURE ON** command. You must have write permission for the file. |
| 103 | **cannot open ESCORT Dump file escort.dp**{*proc-id*} | ESCORT is unable to open the *escort.dp*{*proc-id*} file specified by a **DUMP** command. You must have write permission for the file. |

## Run Time Error Messages

A list of the more common run time error messages follows. Those messages marked *OIA* indicate that the message is displayed in the operator information area. All other run time errors terminate ESCORT, and the message is written to the standard error.

| | Error Message | Description |
|---|---|---|
| | **ESCORT Syntax Errors** | A syntax error occurred, refer to the *escort.pr{proc-id}* file for details of the error. |
| | **Communication Controller Error** | The communication processor is not running. |
| | **Logical Unit Requested does not Exist** | The logical unit environment variable, D3274, contains an invalid logical unit. |
| **OIA** | **INHIBIT BAD KEY TRANSLATION** | An undefined key sequence has been entered. Press RESET to continue. |
| | **Insufficient Memory to run ESCORT** | Memory allocation of 512K required for each ESCORT process.<br>The UNIX tunable parameters need to be changed to allow each UNIX process to run using 512K of memory. Refer to your UNIX System Administrator's User Guide. |
| | **The environment variable TERM is not defined** | ESCORT is unable to set up your terminal for execution. The TERM environment variable must be set before using ESCORT. |
| | **Your terminal is unknown to this system** | Your terminal type is not defined in the UNIX *terminfo* data base. |

| | Error Message | Description |
|---|---|---|
| OIA | **INHIBIT ILLEGAL FUNCTION** | Your application does not accept the function entered. Press RESET to continue. |
| OIA | **SNA/BSC Terminal is busy** | Either the D3274 logical unit requested for the host connection is currently being used by another user, or all logical unit connections assigned to you are in use. |
| OIA | **No lu ports available** | All logical unit connections are in use. |
| OIA | **INHIBIT NOT HERE** | Attempt to either enter data in a protected area, or enter alphabetic data in a numeric field. Press RESET to continue. |
| | **ESCORT Execution Errors** | An error occurred during script execution, refer to the *escort.pr{proc-id}* file for details of the error. |
| OIA | **ASYNC Connection Failed** | A connection to an asynchronous host failed; refer to the *escort.pr{proc-id}* file for details of the error. The *Basic Network Utilities* documentation provides further information on failed asynchronous host connections. |
| OIA | **Cannot open ESCORT ASG ky file** | ESCORT is unable to open the *escort.ky{proc-id}* file specified by Interactive mode Automatic Script Generation. You must have write permission for the file. |
| OIA | **Cannot open ESCORT ASL lg file** | ESCORT is unable to open the *escort.lg{proc-id}* file specified by Interactive mode Automatic Screen Logging. You must have write permission for the file. |

| Error Message | Description |
|---|---|
| **Cannot open ESCORT Print file: escort.pr**{*proc-id*} | The ESCORT process was unable to open the *escort.pr*{*proc-id*} file. You must have write permission for the file. |

# Debugging Facilities

The ESCORT commands, **TRACE** and **DUMP**, are tools designed to assist you in debugging *hard to find* problems in scripts.

This appendix describes the use of these debugging commands, it does not tell you how to analyze their output.

## TRACE Command

Use the **TRACE** command to activate or deactivate the trace facility.

The format of the **TRACE** command is as follows:

```
TRACE (X,   {1})
            {0}
```

X indicates tracing the program execution phase.

The **TRACE** command can be placed anywhere between the **PROG** and **ENDP** statements. The operand **1** toggles tracing on and the operand **0** toggles tracing off. The **TRACE** command can be toggled on and off as required in a script. This allows you either to trace the entire program or to trace portions of the program only.

The output from **TRACE** is directed to a file, created in the directory defined by the ESCDIR environment variable, named *escort.pr*{*proc-id*}, where {*proc-id*} refers to the unique process identification the UNIX operating system assigns to each process.

The format of the execution phase trace line gives you the command code, command mnemonic, and parse list address of each command as it is being executed. Thus, any program loop can be readily found with the execution trace.

### Example 1

In this example, the TRACE command is used to trace the entire program.

```
logoftso prog   main
         trace (X,1)          /* turn tracing on
main     script
         clear
         text    ("logoff")
         enter
         while   !($scan("WELCOME TO"))
         do
                 fresh
         endo
         ends
         endp
```

### Example 2

In this example, the TRACE command is used to trace a portion of program code containing a WHILE loop.

```
logoftso prog   main
main     script
         clear
         text    ("logoff")
         enter
         trace (X,1)          /* turn tracing on
         while   !($scan("WELCOME TO"))
         do
                 fresh
         endo
         trace (X,0)          /* turn tracing off
         ends
         endp
```

## DUMP Command

An ESCORT dump can be produced by using the **DUMP** command. The dump is written to a file, created in the directory defined by the ESCDIR environment variable, named *escort.dp{proc-id}*, where *{proc-id}* refers to the unique process identification the UNIX operating system assigns to each process. A dump is also produced when a program abends. A program abend occurs if there is a user run-time error or a disastrous error in ESCORT itself.

A dump provides you with the following data:

- Current values of all ESCORT table indices.
- Parse list (stored commands and operands) dump.
- Constant and variable tables with table index, storage address, length, type, and value for each entry.
- Label table.
- Script table.
- User file table.
- Format table.
- Static storage area.
- Dynamic storage area.
- Return address stack - nested calls.
- Frame stack - address of storage frames for local variables.

The operands that follow commands stored in the parse list are nearly always represented by the corresponding table indices. Expressions are terminated by hex FFFF. Operators in integer expressions are stored as the two's complement of the ASCII code.
For example, "+" (hex 2B) becomes hex FFD5.

## Example

This example activates **DUMP** after the **ENTER** command is executed.

```
logoftso prog     main
main     script
         clear
         text     ("logoff")
         enter
         dump                /* dump information to
                             /* escort.dump'pid'
         while    !($scan("WELCOME TO"))
         do
                  fresh
         endo
         ends
         endp
```

# AID Subroutines Library

This section provides you with a listing of the scripts for the AID subroutines contained on your ESCORT installation disk. These programs are listed for your information.

See the section, "Synchronous Response/No-Response Mode Transactions", in Chapter 2 for more information about using the AID subroutines.

## AID_GC

Writes a tag in row 24, column 79, sends an AID key, and waits until the tag has been overwritten by the response from the synchronous host system.

```
aid_gc  script  (int aid_key)
        field   (24,79,1) tag_fld
        char    (1) tag

        tag    =  "g"
        tag_fld = tag
        aid     (aid_key)
        while   tag_fld = tag
        do
                fresh
        endo
        ends
```

## AID_CC

Moves the cursor to the last position on the screen, sends an AID key, and waits until the cursor has moved to another location on the screen.

```
aid_cc  script  (int aid_key)

        cursor   (24,80)
        aid     (aid_key)
        while   $getcur = 1920
        do
                fresh
        endo
        ends
```

# AID_01C

Sends an AID key and waits until line 1 changes.

```
aid_01c script  (int aid_key)
        field   (1,1,80) newl_1
        char    (80) oldl_1
        oldl_1  =  newl_1
        aid     (aid_key)
        while   newl_1 = oldl_1
        do
                fresh
        endo
        ends
```

# AID_24C

Sends an AID key and waits until line 24 changes.

```
aid_24c script  (int aid_key)
        field   (24,1,80) newl_24
        char    (80) oldl_24
        oldl_24  =  newl_24
        aid     (aid_key)
        while   newl_24 = oldl_24
        do
                fresh
        endo
        ends
```

# AID_LC

Sends an AID key and waits until data on a specified line has changed.

```
aid_lc  script  (int aid_key, int lcrow)
        int     lcoffset
        char    (80) lcline
        lcoffset = (80*lcrow-79)
        lcline = $gsubstr (SCREEN, lcoffset, 80)
        aid     (aid_key)
        while   $scan (lcline (lcrow, 1, 80))
        do
                fresh
        endo
        ends
```

## AID_FC

Sends an AID key and waits until a specified field on the screen has changed.

```
aid_fc  script  (int aid_key, field (*,*,*) new_fld)
        char    (80) old_fld
        old_fld = new_fld
        aid     (aid_key)
        while   new_fld = old_fld
        do
                fresh
        endo
        ends
```

## AID_SMA

Sends an AID key and waits until a specified message appears on the screen.

```
aid_sma script  (int aid_key, char (*) smsmsg, int smsrow,
                 int smscol, inst smslen)
        aid     (aid_key)
        while   !($scan (smsmsg (smsrow, smscol, smslen)))
        do
                fresh
        endo
        ends
```

## AID_SMD

Sends an AID key and waits until a specified message disappears from the screen.

```
aid_smd script  (int aid_key, char (*) smsmsg, int smsrow,
                 int smscol, inst smslen)
        aid     (aid_key)
        while   $scan (smsmsg (smsrow, smscol, smslen))
        do
                fresh
        endo
        ends
```

## AID_KC

Writes a PF key in row 24, column 74, sends an AID key, and waits until the tag has been overwritten by a response from the synchronous host system.

```
aid_kc  script  (int aid_key)
        field   (24, 74, 5) tag_fld
        char    (5) tag

        switch  (aid_key)
          case 0        tag = "ENTER"
          case 1        tag = " PF1 "
          case 2        tag = " PF2 "
          case 3        tag = " PF3 "
          case 4        tag = " PF4 "
          case 5        tag = " PF5 "
          case 6        tag = " PF6 "
          case 7        tag = " PF7 "
          case 8        tag = " PF8 "
          case 9        tag = " PF9 "
          case 10       tag = " PF10"
          case 11       tag = " PF11"
          case 12       tag = " PF12"
          case 13       tag = " PF13"
          case 14       tag = " PF14"
          case 15       tag = " PF15"
          case 16       tag = " PF16"
          case 17       tag = " PF17"
          case 18       tag = " PF18"
          case 19       tag = " PF19"
          case 20       tag = " PF20"
          case 21       tag = " PF21"
          case 22       tag = " PF22"
          case 23       tag = " PF23"
          case 24       tag = " PF24"
          case 25       tag = "CLEAR"
          case 26       tag = " PA1 "
          case 27       tag = " PA2 "
          case 28       tag = " PA3 "
          case 29       tag = " ATTN"
          case 30       tag = "SYSRQ"
          default       tag = "!!!!!"
        endc
        tag_fld = tag
        aid  (aid_key)
        while tag_fld = tag
        do
             fresh
        endo
        ends
```

## AID_RESP

Moves the cursor to the last position on the screen, sends an AID key, and waits until the cursor has moved to another location on the screen.

This subroutine is used when you press (ESC) **f 0** to activate or deactivate AID subroutine substitution while in Automatic Script Generation.

This is a generic subroutine which you may change to suit your particular application environment.

```
aid_resp script  (int aid_key)

         cursor  (24,80)
         aid     (aid_key)
         while   $getcur = 1920
         do
                 fresh
         endo
         ends
```

# Interpretation of Attribute Bytes

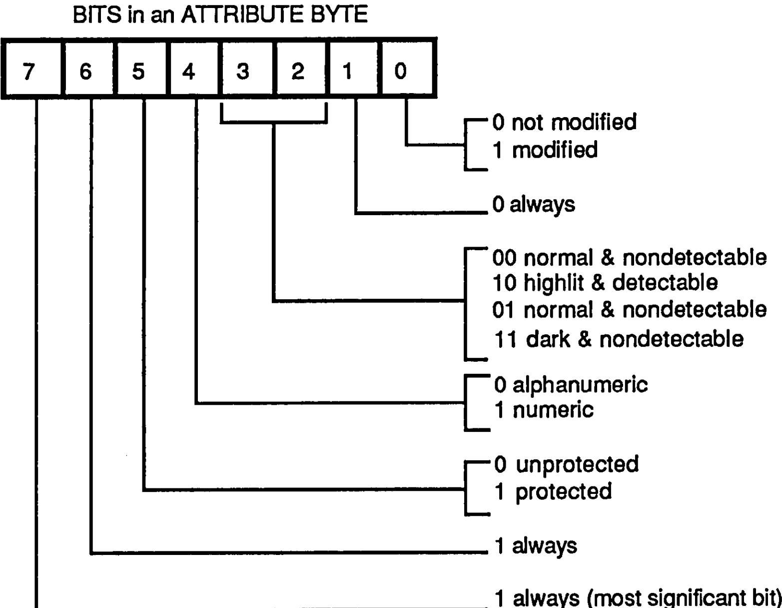
When you press (ESC) **f 5** while connected to a synchronous host session, ESCORT displays an alpha character in the position of each attribute byte on the screen. The character displayed represents the Primary Attributes for each field; this appendix shows you how to interpret the displayed characters.

The following table shows you how to convert the character displayed on the terminal screen into attribute bytes.

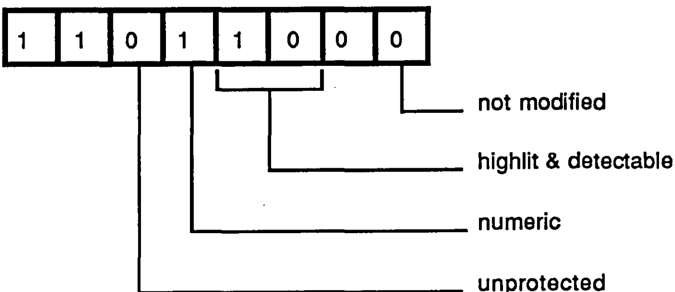
**ASCII Character to Attribute Byte Conversion**

| Char | Attr | Char | Attr | Char | Attr | Char | Attr |
|---|---|---|---|---|---|---|---|
| @ | 1100 0000 | P | 1101 0000 | ' | 1110 0000 | p | 1111 0000 |
| A | 1100 0001 | Q | 1101 0001 | a | 1110 0001 | q | 1111 0001 |
| B | 1100 0010 | R | 1101 0010 | b | 1110 0010 | r | 1111 0010 |
| C | 1100 0011 | S | 1101 0011 | c | 1110 0011 | s | 1111 0011 |
| D | 1100 0100 | T | 1101 0100 | d | 1110 0100 | t | 1111 0100 |
| E | 1100 0101 | U | 1101 0101 | e | 1110 0101 | u | 1111 0101 |
| F | 1100 0110 | V | 1101 0110 | f | 1110 0110 | v | 1111 0110 |
| G | 1100 0111 | W | 1101 0111 | g | 1110 0111 | w | 1111 0111 |
| H | 1100 1000 | X | 1101 1000 | h | 1110 1000 | x | 1111 1000 |
| I | 1100 1001 | Y | 1101 1001 | i | 1110 1001 | y | 1111 1001 |
| J | 1100 1010 | Z | 1101 1010 | j | 1110 1010 | z | 1111 1010 |
| K | 1100 1011 | [ | 1101 1011 | k | 1110 1011 | { | 1111 1011 |
| L | 1100 1100 | \ | 1101 1100 | l | 1110 1100 | \| | 1111 1100 |
| M | 1100 1101 | ] | 1101 1101 | m | 1110 1101 | } | 1111 1101 |
| N | 1100 1110 | ^ | 1101 1110 | n | 1110 1110 | — | 1111 1110 |
| O | 1100 1111 | — | 1101 1111 | o | 1110 1111 | < | 1111 1111 |

The following diagram shows you how to read the bits in an attribute byte. Note that bits 3 and 2 are coupled and are read together. *Detectable* refers to detectable by a light pen.



For example, if the alphabetic character X is displayed when the Display Attribute key sequence ( ESC **f 5**) is pressed, the corresponding attribute byte is 1101 1000. Interpretation of the attribute byte shows that the field has the following characteristics:



# Key Sequences

This appendix lists the key sequences that emulate IBM 3278 key functions for the following terminals:

### Synchronous Terminals

- Standard ASCII terminals
- AT&T 4410 and Teletype® 5410 terminals
- AT&T 4418 and Teletype 5418 terminals
- AT&T 4425 and Teletype 5425 terminals
- AT&T 605 Business Communications Terminal (BCT)
- AT&T 610, 615, 620, and 630 Business Communications Terminals (BCTs).

### Asynchronous Terminals

- DEC VT100 terminal.

Note that certain key functions are ignored if either ESCORT or the terminal does not support them. The following key functions are *not* supported by ESCORT:

| | |
|---|---|
| ALLCAP | COLR |
| ALT_CR | CTRL |
| BLINK | CURSR_SEL |
| BOT | NULLEND |
| CAN | NUM_OV |
| CLICK | TOP |

The synchronous keyboard files are the same as the keyboard files used by the AT&T 3270 Emulator+ software.

## Key Sequences for Standard ASCII Terminals

The AT&T 4415, 5420 and the Tektronix™ 4105 terminals use these key sequences.

| 3278 Key Function | Standard ASCII Terminal Key Sequence |
|---|---|
| ATTN | ESC **a** RETURN |
| BAKTAB | CTRL **e** |
| BS | CTRL **h** |
| CENT | ^ |
| CLEAR | ESC **z** |
| DEL | DEL |
| DEV_CNCL | ESC **d** |
| DOWN_A | CTRL **v** |
| DUP | CTRL **d** |
| E_EOF | ESC **e f** |
| E_INPUT | ESC **e i** |
| ENTER | RETURN |
| ENTER1 | RETURN |
| EXIT | ESC **x x** |
| FM | CTRL **k** |
| HOME | CTRL **o** |
| IDENT | ESC **i** |
| INS | CTRL **u** |
| LDUB | CTRL **r** |
| LEFT_A | CTRL **f** |
| NEWL | CTRL **j** |
| NEXTS | ESC + |
| NOT | ] |
| PA1 to PA3 | ESC **a** *key number* RETURN |
| PF1 to PF24 | ESC *key number* RETURN |
| PREVS | ESC − |
| PRINT | ESC **p** |
| RDUB | CTRL **y** |
| REDRAW | ESC **r** |
| RESET | CTRL **a** |
| RIGHT_A | CTRL **g** |

## Key Sequences for Standard ASCII Terminals (continued)

| 3278 Key Function | Standard ASCII Terminal Key Sequence |
|---|---|
| SHELL | ESC s |
| SOLID | [ |
| STAT | ESC l |
| SYS_REQ (SNA only) | ESC q |
| TAB | CTRL i |
| TEST_REQ (BSC only) | ESC q |
| UP_A | CTRL t |

# Key Sequences for AT&T 4410 and Teletype 5410 Terminals

| 3278 Key Function | AT&T 4410 and Teletype 5410 Key Sequence |
|---|---|
| ATTN | [ESC] **a** [RETURN] |
| BAKTAB | [↙] |
| BS | [CTRL] ^ **h** |
| CENT | ^ |
| CLEAR | [ESC] **z** |
| DEL | [DEL] |
| DEV_CNCL | [ESC] **d** |
| DOWN_A | [↓] |
| DUP | [CTRL] **d** |
| E_EOF | [ESC] **e f** |
| E_INPUT | [ESC] **e i** |
| ENTER | [RETURN] |
| ENTER1 | [RETURN] |
| EXIT | [ESC] **x x** |
| FM | [CTRL] **k** |
| HOME | [↖] |
| IDENT | [ESC] **i** |
| INS | [CTRL] **u** |
| LDUB | [CTRL] **r** |
| LEFT_A | [←] |
| NEWL | [CTRL] **j** |
| NEXTS | [ESC] > |
| NOT | ] |
| PA1 to PA3 | [ESC] **a** *key number* [RETURN] |
| PF1 to PF9 | [ESC] *key number* |
| PF10 | [ESC] **0** |
| PF11 | [ESC] – |
| PF12 | [ESC] = |
| PF13 | [ESC] [SHIFT] **1** |
| PF14 | [ESC] [SHIFT] **2** |
| PF15 | [ESC] [SHIFT] **3** |

## Key Sequences for AT&T 4410 and Teletype 5410 Terminals (continued)

| 3278 Key Function | AT&T 4410 and Teletype 5410 Key Sequence |
|---|---|
| PF16 | ESC SHIFT 4 |
| PF17 | ESC SHIFT 5 |
| PF18 | ESC SHIFT 6 |
| PF19 | ESC SHIFT 7 |
| PF20 | ESC SHIFT 8 |
| PF21 | ESC SHIFT 9 |
| PF22 | ESC SHIFT 0 |
| PF23 | ESC SHIFT – |
| PF24 | ESC SHIFT = |
| PREVS | ESC < |
| PRINT | ESC p |
| RDUB | CTRL y |
| REDRAW | ESC r |
| RESET | CTRL a |
| RIGHT_A | → |
| SHELL | ESC s |
| SOLID | [ |
| STAT | ESC l |
| SYS_REQ (SNA only) | ESC q |
| TAB | CTRL i |
| TEST_REQ (BSC only) | ESC q |
| UP_A | ↑ |

## Key Sequences for AT&T 4418 and Teletype 5418 Terminals

Note that on these terminals there is no key marked ESC or CTRL.

To emulate ESC, press ALT and |.

For CTRL use the key immediately to the left of the space bar.

| 3278 Key Function | AT&T 4418 and Teletype 5418 Key Sequence |
|---|---|
| ATTN | ATTN |
| BAKTAB | BACK TAB |
| BS | BACK SPACE |
| CENT | ^ |
| CLEAR | CLEAR |
| DEL | DEL |
| DEV_CNCL | DEV CNCL |
| DOWN_A | ↓ |
| DUP | DUP |
| E_EOF | ERASE EOF |
| E_INPUT | ERASE INPUT |
| ENTER | ENTER (lower right side) |
| ENTER1 | ENTER (upper left side) |
| EXIT | ESC x x |
| FM | FIELD MARK |
| HOME | HOME |
| IDENT | IDENT |
| INS | INS |
| LDUB | ← ← |
| LEFT_A | ← |
| NEWL | NEW LINE |
| NEXTS | ESC + |
| NOT | ] |

## Key Sequences for AT&T 4418 and Teletype 5418 Terminals (continued)

| 3278 Key Function | AT&T 4418 and Teletype 5418 Key Sequence |
|---|---|
| PA1 | PA1 |
| PA2 | PA2 |
| PA3 | SHIFT INS |
| PF1 to PF24 | PF1 to PF24 |
| PREVS | ESC – |
| PRINT | PRINT LCL |
| RDUB | → → |
| REDRAW | ESC **r** |
| RESET | RESET (lower left side) |
| RIGHT_A | → |
| SHELL | SHIFT RESET (upper left side) |
| SOLID | [ |
| STAT | RESET (upper left side) |
| SYS_REQ (SNA only) | SYS REQ |
| TAB | CURSOR TAB |
| TEST_REQ (BSC only) | SYS REQ |
| UP_A | ↑ |

# Key Sequences for AT&T 4425 and Teletype 5425 Terminals

| 3278 Key Function | AT&T 4425 and Teletype 5425 Key Sequence |
|---|---|
| ATTN | ESC **a** RETURN |
| BAKTAB | SHIFT TAB |
| BS | BACK SPACE |
| CENT | ^ |
| CLEAR | CLEAR |
| DEL | DEL |
| DEV_CNCL | ESC **d** |
| DOWN_A | ↓ |
| DUP | CTRL **d** |
| E_EOF | CLEAR LINE |
| E_INPUT | DELETE LINE |
| ENTER | RETURN |
| ENTER1 | ENTER (on keypad) |
| EXIT | ESC **x x** |
| FM | CTRL **k** |
| HOME | HOME |
| IDENT | ESC **i** |
| INS | INSERT CHAR |
| LDUB | CTRL **r** |
| LEFT_A | ← |
| NEWL | CTRL **j** |
| NEXTS | ESC + |
| NOT | ] |
| PA1 to PA3 | ESC **a** *key number* RETURN |
| PF1 to PF4 | PF1 to PF4 |
| PF5 | **7** (on keypad) |
| PF6 | **8** (on keypad) |
| PF7 | **9** (on keypad) |
| PF8 | − (on keypad) |
| PF9 | **4** (on keypad) |
| PF10 | **5** (on keypad) |

## Key Sequences for AT&T 4425 and Teletype 5425 Terminals (continued)

| 3278 Key Function | AT&T 4425 and Teletype 5425 Key Sequence |
|---|---|
| PF11 | **6** (on keypad) |
| PF12 | **,** (on keypad) |
| PF13 to PF24 | ESC *key number* RETURN |
| PREVS | ESC **–** |
| PRINT | ESC **p** |
| RDUB | CTRL **y** |
| REDRAW | ESC **r** |
| RESET | CTRL **a** |
| RIGHT_A | → |
| SHELL | ESC **s** |
| SOLID | **[** |
| STAT | ESC **l** |
| SYS_REQ (SNA only) | ESC **q** |
| TAB | TAB |
| TEST_REQ (BSC only) | ESC **q** |
| UP_A | ↑ |

## Key Sequences for AT&T 605 Business Communications Terminal

The AT&T 605 Business Communications Terminal (BCT) has a 102-key keyboard.

| 3278 Key Function | AT&T 605 BCT Key Sequence |
|---|---|
| ATTN | ESC a RETURN |
| BAKTAB | SHIFT TAB |
| BS | BACK SPACE |
| CENT | ^ |
| CLEAR | SHIFT CLEAR |
| DEL | CTRL DELETE |
| DEV_CNCL | ESC d |
| DOWN_A | ↓ |
| DUP | CTRL d |
| E_EOF | ESC e f |
| E_INPUT | SHIFT DEL LN |
| ENTER | RETURN |
| ENTER1 | RETURN |
| EXIT | ESC x x |
| FM | CTRL k |
| HOME | CLEAR HOME |
| IDENT | ESC i |
| INS | INS LN |
| LDUB | CTRL r |
| LEFT_A | ← |
| NEWL | CTRL j |
| NEXTS | ESC + |
| NOT | ] |
| PA1 to PA3 | ESC a *key number* RETURN |
| PF1 to PF24 | ESC *key number* RETURN |
| PREVS | ESC – |
| PRINT | ESC p |
| RDUB | CTRL y |
| REDRAW | ESC r |
| RESET | ESC c |
| RIGHT_A | → |

## Key Sequences for AT&T 605 Business Communications Terminal (continued)

| 3278 Key Function | AT&T 605 BCT Key Sequence |
|---|---|
| SHELL | ESC s |
| SOLID | [ |
| STAT | ESC l |
| SYS_REQ (SNA only) | ESC q |
| TAB | TAB |
| TEST_REQ (BSC only) | ESC q |
| UP_A | ↑ |

# Key Sequences for AT&T 610, 615, 620, and 630 Business Communications Terminals

The AT&T 610, 615, 620, and 630 Business Communications Terminals (BCTs) have 98-key keyboards.

| 3278 Key Function | AT&T 610, 615, 620, and 630 BCTs Key Sequence |
|---|---|
| ATTN | ESC **a** RETURN |
| BAKTAB | SHIFT TAB |
| BS | BACK SPACE |
| CENT | ^ |
| CLEAR | CLEAR |
| DEL | DELETE |
| DEV_CNCL | ESC **d** |
| DOWN_A | ↓ |
| DUP | CTRL **d** |
| E_EOF | ESC **e f** |
| E_INPUT | ESC **e i** |
| ENTER | RETURN |
| ENTER1 | RETURN |
| EXIT | ESC **x x** |
| FM | CTRL **k** |
| HOME | HOME |
| IDENT | ESC **i** |
| INS | CTRL **u** |
| LDUB | CTRL **r** |
| LEFT_A | ← |
| NEWL | CTRL **j** |
| NEXTS | ESC **+** |
| NOT | ] |
| PA1 to PA3 | ESC **a** *key number* RETURN |
| PF1 to PF24 | ESC *key number* RETURN |
| PREVS | ESC **–** |
| PRINT | ESC **p** |
| RDUB | CTRL **y** |
| REDRAW | ESC **r** |
| RESET | ESC **c** |
| RIGHT_A | → |

## Key Sequences for AT&T 610, 615, 620, and 630 Business Communications Terminals (continued)

| 3278 Key Function | AT&T 610, 615, 620, and 630 BCTs Key Sequence |
|---|---|
| SHELL | ESC s |
| SOLID | [ |
| STAT | ESC l |
| SYS_REQ (SNA only) | ESC q |
| TAB | TAB |
| TEST_REQ (BSC only) | ESC q |
| UP_A | ↑ |

## Key Sequences for DEC VT100 Terminal

| DEC VT100 Key Function | ESCORT Equivalent Key Sequence |
|---|---|
| BAKTAB | CTRL **e** |
| BS | CTRL **h** |
| CLEAR | ESC **z** |
| DEL | DEL |
| DOWN_A | CTRL **v** |
| ENTER | RETURN |
| ENTER1 | RETURN |
| EXIT | ESC **x x** |
| HOME | CTRL **o** |
| IDENT | ESC **i** |
| LEFT_A | CTRL **f** |
| NEWL | CTRL **j** |
| NEXTS | ESC **+** |
| PF1 to PF8 | ESC *key number* RETURN |
| PREVS | ESC **–** |
| PRINT | ESC **p** |
| REDRAW | ESC **r** |
| RIGHT_A | CTRL **g** |
| SHELL | ESC **s** |
| STAT | ESC **l** |
| TAB | CTRL **i** |
| UP_A | CTRL **t** |

**Note**
This key sequence table should be used in asynchronous-only environments. If you communicate with both synchronous and asynchronous hosts, use the appropriate key sequence table for your specific synchronous terminal for all synchronous and asynchronous applications.

# Environment Variables and Customization

This appendix provides information on setting environment variables applicable to your operating environment and on customizing terminal functions for different types of ASCII terminals.

# Setting 3270 Emulator+ ESCORT Environment Variables

Once the ESCORT software has been installed, certain prerequisite variables should be set in your *profile* file. When invoking ESCORT, ensure that the 3270 Emulator+ terminal manager process is running.

## Terminal Environment Variable

ESCORT uses the environment variable, *TERM*, to access terminal information in the system file *terminfo* for screen management. The following example shows the environment variable set for an AT&T 4410 terminal type.

```
TERM=4410
export TERM
```

## 3270 Emulator+ Environment Variables

ESCORT runs in conjunction with the AT&T 3270 Emulator+ software. Your *profile* file should be edited to include the following command:

```
. /usr/bscadm/runtime/bscenvset
```

or

```
. /usr/snaadm/runtime/snaenvset
```

Set the appropriate environment variables so that 3270 Emulator+ and ESCORT will execute properly.

## D3274 Environment Variable

The default value for the D3274 environment variable provided by the *snaenvset* command allows access to all available logical unit connections. Setting the D3274 environment variable provides controlled access to certain host applications. You can assign ranges of logical unit ports to particular users. In the following example, a user is given access to eight logical unit ports.

```
D3274=1-5,13,14,15
export D3274
```

## Host/Local Session Environment Variable

The UNIX operating system environment variable, *ESCHOST*, determines whether a synchronous connection is to be established. The environment variable can be set to 0 or 1; if set to a value of 1 (the default value if this variable is not set) the ESCORT script will be able to connect to a synchronous host session.

Setting the ESCHOST environment variable to 0 is useful

- in limiting access to prototyping local screen formats
- if ISC or SDLI cards have not been installed in the 3B processor
- if the user accesses only asynchronous host applications.

In the following example, a user's ESCORT connections will default to a local session.

```
ESCHOST=0
export ESCHOST
```

## Directory Environment Variable

The UNIX operating system environment variable, *ESCDIR*, determines the path for the five types of ESCORT utility files. If the *ESCDIR* environment variable is not set, ESCORT utility files are created in your *$HOME* directory. In the following example, a user's ESCORT utility files will be created in a directory named *sys_1*, a subdirectory of */usr/john*.

```
ESCDIR=/usr/john/sys_1
export ESCDIR
```

## Terminal Information Environment Variable

To use the *terminfo* terminal information files installed by ESCORT, set the *terminfo* variable as follows:

```
TERMINFO=/usr/escort/terminfo
export TERMINFO
```

Setting this environment variable is only necessary if the system-supplied files contain errors or have been modified in some way and ESCORT does not function correctly.

## Terminal Customization

The screen and keyboard layouts of various types of ASCII terminals differ from those found on actual IBM 3278 display stations. The AT&T 3270 Emulator+ software is designed to work with many different types of ASCII terminal by using a terminal emulator process to translate the logical IBM 3278 functions to the target ASCII terminal.

To be consistent, ESCORT uses the same keyboard sequence defined in the AT&T 3270 Emulator+ software. The AT&T 3270 Emulator+ key sequences that are supported by ESCORT, together with the default ESCORT specific keys, are listed in this section.

You should utilize the AT&T 3270 Emulator+ software utilities *kyinit* and *scinit* to customize the IBM 3278 functions. Follow the procedure outlined in the *AT&T 3270 Emulator+ User's and System Administrator's Guides.*

### Caution

If you modify the keyboard source files and fail to run the *kyinit* utility, it is possible that the key sequences generated will not be unique.

The default ESCORT specific key sequences can be modified for your particular environment by appending the ESCORT key labels and their associated values to the keyboard mapping files in the AT&T 3270 Emulator+ software. The following table details the default values for the ESCORT specific keys.

| Default Values of ESCORT Specific Keys |
|---|
| :EK_F0=\Ef0:\ |
| :EK_F1=\Ef1:\ |
| :EK_F2=\Ef2:\ |
| :EK_F3=\Ef3:\ |
| :EK_F4=\Ef4:\ |
| :EK_F5=\Ef5:\ |
| :EK_F6=\Ef6:\ |
| :EK_F7=\Ef7:\ |
| :EK_F9=\Ef9:\ |
| :EK_OPENS=\Eos:\ |
| :EK_OPENA=\Eoa:\ |

## Defining Multiple Key Sequences

As part of the terminal customization feature, ESCORT allows you to specify two separate key sequences for the same function. For example, when customizing a standard ASCII terminal you may wish to specify the key sequence ESC **Z**, in addition to the standard key sequence of ESC **z**, to represent the function, CLEAR.

In this case, the mapping file should be amended to included the following:

```
KY_CLEAR=\Ez:\
KY_CLEAR=\EZ:\
```

| AT&T 3270 Emulator+ Supported Keys | |
|---|---|
| ATTN | LEFT_A |
| BAKTAB | NEWL |
| BS | NEXTS |
| CENT | NOT |
| CLEAR | PA1 to PA3 |
| DEL | PF1 to PF24 |
| DEV_CNCL | PREVS |
| DOWN_A | PRINT |
| DUP | RDUB |
| E_EOF | REDRAW |
| E_INPUT | RESET |
| ENTER | RIGHT_A |
| ENTER1 | SHELL |
| EXIT | SOLID |
| FM | STAT |
| HOME | SYS_REQ |
| IDENT | TAB |
| INS | TEST_REQ |
| LDUB | UP_A |

| ESCORT Specific Keys | |
|---|---|
| Function | Key Sequence |
| QUIT | ESC **f 1** |
| I/R | ESC **f 2** |
| ASG | ESC **f 3** |
| CURSR_POS | ESC **f 4** |
| ATTRIB | ESC **f 5** |
| ASL | ESC **f 6** |
| KEY_STATUS | ESC **f 7** |
| SHOW | ESC **f 9** |
| AID_SUB | ESC **f 0** |
| OPENS | ESC **o s** |
| OPENA | ESC **o a** |

# Additional Programs

This section contains more advanced programs written in ESCORT for more experienced programmers to use.

These programs serve two purposes. They provide you with scripts that you may be able to modify for use with your particular application, and they give you an idea of how to write more complicated programs in ESCORT.

A short description before each program listing explains what the program does and points out any important programming techniques used.

## Writing a Tutorial Script

This program can be used as a model for writing a tutorial script. The program accesses a sample host application and must be modified to suit your particular application.

It employs a subroutine that takes a set of literals to be entered by an operator in a training session and displays them in a window. When the operator has entered the data, the subroutine checks the data at locations passed in the global array *offsets*. If the data entered is not what was requested, an error message is displayed and the operator must reenter the data.

```
tutor    prog main
         int i
         int j
         int k
         int l
         int rc
         int offsets (12)
         char (20) values (12)

main     script

         clear
         tab
         offsets (1) = 855              /* where to check for "imstest"
         values = ("imstest", "end")
         call check                     /* won't return until "imstest"
         enter                          /* has been entered

         offsets = (505, 825)
         values = ("userid", "imsgrp", "end")
         call check
         enter

         offsets = (1613)
         values = ("/test mfs", "end")
         call check
         enter

repeat:
         window (1,63,6,80)
         wto "Hit CLEAR key"
         exit (tutorial)
         if sysaid != 25 then           /* operator did not hit CLEAR
           window (9,63,13,80)
           j = (j + 20)
           wto ("Wrong, again !  That's a $ " j " fine.")
           goto repeat                  /* loop
         endif
         clear                          /* send the CLEAR

         offsets = (1)
         values = ("/rcl", "end")
         call check
         enter                          /* sign off

         ends
```

```
check     script

          window (1,63,6,80,r)
          wto ("Enter:")
          for i = 1 to 100                /* display values in window
            do
              if values (i) = "end" then
                break
              endif
              wto values (i)
            endo
          exit (tutorial)
again:
          rc = 0
          for i = 1 to 100
            do
              if values (i) = "end" then
                break
              endif
              l = $length (values (i))
              k =  offsets (i)
                                          /* check for correct data
              if $gsubstr (screen, k, l) != values (i) then
                rc = 1
              endif
            endo
         if rc = 1 then                   /* error
           erin
           window (1,63,6,80)
           wto ("Enter:")
           for i = 1 to 100
             do
               if values (i) = "end" then
                 break
               endif
               wto values (i)
             endo
           window (9,63,13,80)
           j = (j + 20)
           wto ("Wrong, again !  That's  a $ " j " fine.")
           exit (tutorial)
           goto again
         endif
         ends
         endp
```

## Performing Regression Testing

This program performs regression testing on the PF1 (Find) key in an order entry application. Numerous comments are included to guide you through the program. You may be able to use this program for your synchronous host application with some slight modifications.

The program consists of 2 scripts. The main script logs on to an application, brings up a particular order entry screen, and then accesses a second script that performs a regression test on the PF1 (Find) key. Note the use of the AID subroutines and the **LOG** command to save the results of the test.

```
ORDERS   PROG   MAIN

         /***Global Variable Declaration

         copy  "c:\mfs3\orders"       /* get format variables
         field (24,55,20)  action       /* field to indicate pfkey hit
         char  (8)      mfs             /* MFS name of the order screen

         char  (9)      goodord         /* valid order number
         char  (9)      badord          /* invalid order number

         char  (3)      goodcid1        /* valid customer id - part1
         char  (3)      goodcid2        /* valid customer id - part2
         char  (4)      goodcid3        /* valid customer id - part3

         char  (3)      badcid1         /* invalid customer id - part1
         char  (3)      badcid2         /* invalid customer id - part2
         char  (4)      badcid3         /* invalid customer id - part3

MAIN     SCRIPT

         /***Local Variables Declaration

         int   rtncode                  /* return code
         int   findflag                 /* results of find test

         /***Initialize Global Variables

         mfs     = "orders"             /* save name of screen
         goodord = "15981"              /* valid order number
         badord  = "16601"              /* invalid order number

         goodcid1 = "000"               /* valid customer id
         goodcid2 = "004"               /* valid customer id
         goodcid3 = "9411"              /* valid customer id

         badcid1  = "000"               /* invalid customer id
         badcid2  = "004"               /* invalid customer id
         badcid3  = "4818"              /* invalid customer id

         /***Logon to Application
```

```
        call   logon (rtncode)          /* logon to application
        if  (rtncode != 0)              /* is there a mistake?
        then
            abend                       /* couldn't logon, so quit
        endif

        /***Call Up Screen

        call  aid_resp (25)             /* clear screen
        text  ("/for orders")           /* request orders format
        call  aid_resp (0)              /* press enter to bring up screen

        /***Do Find Key Regression Test

        call  pf1reg (rtncode)          /* run find key regression test
        findflag = rtncode              /* save the results

        /***Logoff Application

        call aid_resp (25)              /* clear screen
        text  ("/rcl")                  /* tells system we want to logoff
        call aid_resp (0)               /* press enter to end session

        ENDS

PF1REG  SCRIPT  (int rtncode)

        /***Initialize Variables

        char (80) message               /* error message
        rtncode = 0                     /* assume good return code

        /***Establish Format

        format  orders                  /* set format to orders

        /***Do Find With Valid Data

        .orderno = goodord              /* load good order
        action = " ACTION = FIND "      /* tell user which pfkey pressed
        call aid_gc (1)                 /* press find key
        if !($scan("FIND COMPLETED"))   /* check for error
        then
           rtncode = (rtncode + 4)      /* set bad return code
           message = "PF1 - GOOD KEY TEST FAILED"
           call error (message,mfs)     /* handle the error
        endif
        action = " ACTION = REFRESH "   /* tell user which pfkey pressed
        call aid_gc(8)                  /* refresh the screen

        /***Do Find With Zeroes

        .orderno = "0000000000"         /* load zero order
        action = " ACTION = FIND "      /* tell user which pfkey pressed
        call aid_gc (1)                 /* press find key
        if !($scan("INVALID ORDER SEGMENT NUMBER"))  /* check for error
        then
           rtncode = (rtncode + 2)      /* set bad return code
           message = "PF1 - ZERO KEY TEST FAILED"
           call error (message,mfs)     /* handle the error
        endif
        action = " ACTION = REFRESH "   /* tell user which pfkey pressed
        call aid_gc(8)                  /* refresh the screen
```

```
/***Do Find With Invalid Data

.orderno = badord                 /* load bad order
action = " ACTION = FIND "        /* tell user which pfkey pressed
call aid_gc (1)                   /* press find key
if !($scan("SECURITY VIOLATION")) /* check for error
then
   rtncode = (rtncode + 1)        /* set bad return code
   message = "PF1 - BAD KEY TEST FAILED"
   call error (message,mfs)       /* handle the error
endif
action = " ACTION = REFRESH "     /* tell user which pfkey pressed
call aid_gc(8)                    /* refresh the screen

/***Log Completion of PF1 Regression Test

log ("ORDERS - PF1 - REGRESSION TEST COMPLETED")

if  (rtncode = 0)                 /* check for all good runs
then
    log ("ORDERS - PF1 - NO ERRORS FOUND")
endif

ENDS
ENDP
```

## Reading from a Pipe File

This program demonstrates the ability of ESCORT to allow a user to read data from a file opened as a pipe. The script complements the "Writing to a Pipe File" program detailed in this section.

The script uses the **WAIT** command to ignore the end-of-file condition that would arise if the pipe file is read before data has been written. A true end-of-file flag must be agreed upon beforehand within the reading and writing scripts; in this example, the variable *buffer* will contain the flag *STOP* indicating that no more records exist.

### Note

The file used in this program must first have been created as a named pipe using the UNIX *mknod* system call.

```
rpipe     prog      main
main      script

          char (80)  buffer

          open(pipe,"/usr/myname/testpipe",R)  /* open file for read
          if sysret = -1                       /* test for failed open
          then
            log("OPEN FAILED")
          endif

          while(1)
          do
            read(pipe,buffer)                 /* read record from pipe
            if sysret = -1                     /* no data in pipe, wait
            then
              wait(30)
              cycle
            endif
            if buffer = "STOP"                 /* no more records, exit
            then
              close(pipe)
              return
            endif
          .                                   /* process record
          .
          endo

          ends
          endp
```

## Writing to a Pipe File

This program demonstrates the ability of ESCORT to allow a user to write data to a file opened as a pipe. The script complements the "Reading from a Pipe File" program detailed in this section.

A true end-of-file flag must be agreed upon beforehand within the reading and writing scripts since the reading script will ignore the usual end-of-file condition indicated by the system global variable, *SYSRET*. In this example, the variable *buffer* will contain the flag *STOP* indicating that no more records exist.

### Note

The file used in this program must first have been created as a named pipe using the UNIX *mknod* system call.

```
wpipe      prog       main
main       script

           char (80)  buffer
           char (20)  usersays

           open(pipe,"/usr/myname/testpipe",W)  /* open file for write
           if sysret = -1                       /* test for failed open
           then
             log("OPEN FAILED")
           endif

           while(1)                            /* process user requests
           do
             if usersays = "STOP"              /* no more records
             then
               buffer = "STOP"                 /* notify reading process
                                               /* no more records
             else
             .                                 /* build record in buffer
             .
             endif
           write(pipe,buffer)                  /* write record to pipe
           if sysret = -1                       /* test for failed write
           then
             log("WRITE FAILED")
           endif
           endo

           ends
           endp
```

# Glossary

This glossary contains definitions for terms and acronyms used throughout this guide. These terms are defined according to their meaning in ESCORT and may not have the same meaning in other programming languages.

**Active session** — Any host or local session connected to a script or connected interactively, to which all ESCORT commands are directed.

**Administrative command** — A command that shows where a program, subroutine, or comment begins or ends. **PROG** is an administrative command.

**AID key** — Attention-identifier key. AID key commands simulate the action of one of the attention-identifier keys. ENTER is an example of an AID key.

**Arithmetic operator** — A character (such as +) that represents a mathematical operation.

**Array** — A collection of values of the same type referred to by a single name. Each entry in an array is called an element.

**Automatic screen logging** — A feature of ESCORT that saves the image of a specific application screen and any data entered on it. Also called ASL.

**Automatic script generation** A feature of ESCORT that automatically creates a script from a user's terminal session. Also called ASG.

**Concatenation** The operation that joins two strings together.

**Constant** A fixed value or data item. A constant may be a string or a numeric constant.

**Debugging command** A programming aid used to check for errors or to detect failures in program execution. **DUMP** is a debugging command.

**Declaration** A statement that defines the type and amount of data associated with a symbolic label. Declaration commands include **INT** and **CHAR**.

**Default** The value or option that is assumed when none is given. For example, if you use the command **BTAB** and do not specify the number of back-tabs to be executed, ESCORT assumes the value is 1.

**Destination** The variable to the left of the equal sign in an assignment statement.

**Emulator** The ESCORT component that allows an ASCII terminal to perform the functions of an IBM 3278 terminal in the synchronous environment, or a DEC VT100 terminal in the asynchronous environment.

**Expression** A single operand or multiple operands separated by operators.

**Extended Field Attributes**
Three arguments used in a **FIELD** statement that define advanced screen characteristics similar to those found in IBM synchronous host screen formats. Also called EFA.

**Field variable**
An area of the screen buffer defined by the user and assigned a symbolic name. Also called screen field variable.

**File management command**
A command used to open or close a file or to control the input and output of data from a file. **READ** is a file management command.

**Format**
A symbolic name for a group of fields that constitute a screen.

**Function**
An algorithm that returns an integer or string value. Function names in ESCORT are preceded by a $.

**Global variable**
A variable that is accessible throughout a program.

**Host session**
The connection of your ASCII terminal through the 3B processor to a synchronous or an asynchronous host computer, providing you with access to a host computer application.

**ISC card**
Intelligent Serial Controller card. A card installed in the 3B2 processor that provides communications ability to synchronous host computers.

**Interactive mode** — An ESCORT feature that allows you to use your ASCII terminal as a synchronous IBM 3278 terminal, or an asynchronous DEC VT100 terminal.

**Interpreter** — The ESCORT component that executes a script.

**Keyword** — An operand predefined by ESCORT, such as *SCREEN*.

**Label** — A word or symbol used at the beginning of a program statement to branch from a **GOTO** statement.

**Local screen format** — Templates used to provide an interface between you and host computer applications.

**Local session** — The connection of your ASCII terminal to the 3B processor, allowing access to user defined screen layouts (see *Local screen format*) that provide an interface between the user and host applications, and allows the user to exchange data between applications.

**Local variable** — A variable that is defined only for a particular script or subprogram.

**Loop** — A series of instructions that is executed repeatedly until a given condition is met.

**Null string** — A string of zero length (with no characters). It is represented by two double quotation marks ("").

**Offset** — The number of bytes from a starting point to some other point.

**Operand** — A constant or variable that is acted on by an operator. *Operand* also refers to any argument that follows a command or function.

**Operator** — A character that represents a mathematical (or logical) operation. ESCORT uses arithmetic, relational, and string concatenation operators.

**Operator information area** — An area on the terminal display screen, in which messages to the operator are written by ESCORT. On terminals with 25-line screens, this area is on line 25, extending from column 21 to column 80. On terminals with 24-line screens, this area is on line 24, extending from column 21 to column 80. Also called *Status Line*.

**Operator notification command** — A command used to communicate with the operator of a terminal. **WTO** is an operator notification command.

**Overflow** — An error that develops when the value returned by an operation is too large for a given register or location.

**Parameter** — An argument passed to a subroutine on a **CALL** statement.

| | |
|---|---|
| **Parser** | The ESCORT component that decodes a program and checks syntax. |
| **Preprocessor command** | A command that requests an action before program execution. **COPY** is a preprocessor command. |
| **Presentation space** | See *Screen buffers*. |
| **Primary Attributes** | Four arguments used in a **FIELD** statement that define basic screen characteristics similar to those found in IBM synchronous host screen formats. |
| **Program control command** | A command that changes the path of program execution. **BREAK** is a program control command. |
| **Relational operator** | A character (such as =) that represents a comparison of two values. |
| **Reserved word** | A word used in ESCORT for a special purpose, such as a command or function name. Reserved words cannot be used as names for variables, labels, programs, or scripts. |
| **SDLI card** | Synchronous Data Link Interface card. A card installed in the 3B5 or 3B15 processor that provides synchronous communications ability between host computers and terminals. |

**Screen buffers** Buffers maintained by ESCORT which contain the last refreshed screen image for each host and local session. The active session screen buffer can be accessed with the system global variable named *SCREEN*. Also called *Presentation space*.

**Screen field variable** See *Field variable*.

**Script** A program written in ESCORT. *Script* also means a subroutine labeled with a script name.

**Script mode** This mode executes an ESCORT script.

**Simulator** The ESCORT component that consists of a parser and an interpreter. The simulator checks for correct syntax and executes the parsed code.

**Statement** An instruction to the computer to perform some sequence of operations. A statement consists of a command or function and its operands.

**Status line** See *Operator information area*.

**String** A sequence of characters or words. ESCORT uses string constants, string variables, and string array variables. String constants are enclosed in double quotation marks. String variables are declared in a **CHAR** statement, such as *CHAR (20) name*.

| | |
|---|---|
| **String concatenation operator** | A symbol that links a series of string operands. The plus sign (+) is used as a string concatenation operator in ESCORT. |
| **Terminal keyboard command** | A command that simulates the action of a given keyboard function. For example, **CLEAR** is a terminal keyboard command. |
| **Tutorial mode** | This ESCORT feature provides the ability to perform edit checks on data entered before sending data to the host. It also allows you to set up on-line training sessions. |
| **Variable** | A symbol used to represent a value. There are five types of variables in ESCORT: integer, integer array, string, string array, and field. |
| **Window** | A display area on a screen. Windows are defined in ESCORT by a **WINDOW** statement. |

# Index


## A

ABEND command, 4-8
AID command, 4-9
AID keys, 2-46
AID subroutines, 2-43
  library, C-1
Arithmetic operators, 2-22
ASSIGN (=) command, 4-11
Asynchronous communication
  port initialization, 2-47
Asynchronous host,
  Automatic Script Generation, 2-50
  scanning data, 2-49
  system prompts, 2-49
ATTN command, 4-16
$ATTR function, 4-141
Attribute bytes, interpretation of, D-1

## B

BEEP command, 4-17
BEGFMT/ENDFMT command,
  2-29, 4-18
BREAK command, 4-20
BTAB command, 4-21

## C

CALL command, 4-22
CAPTURE ON/OFF command, 4-26
CHAR command, 2-16, 4-28
Character set, 2-7
$CHDATE function, 4-143
CHGATTR command, 4-30
CHKPT command, 4-32
CLEAR command, 4-34
CLOSE command, 4-35
COLOR command, 4-36
Command summary table, 4-5
Commands,
  ABEND, 4-8
  AID, 4-9
  ASSIGN (=), 4-11
  ATTN, 4-16
  BEEP, 4-17
  BEGFMT/ENDFMT, 2-29, 4-18
  BREAK, 4-20
  BTAB, 4-21
  CALL, 4-22
  CAPTURE ON/OFF, 4-26
  CHAR, 2-17, 4-28
  CHGATTR, 4-30
  CHKPT, 4-32
  CLEAR, 4-34
  CLOSE, 4-35
  COLOR, 4-36
  COMMENT (/*), 4-38
  CONNECT, 4-39
  COPY, 4-43
  CURSOR, 4-45
  CYCLE, 4-46
  DEL, 4-47
  DISCON, 4-48
  DUMP, B-3
  DUP, 4-50
  EJECT, 4-51
  ENDFMT, 2-29
  ENDP, 2-3, 4-52
  ENDS, 2-3, 4-53
  ENTER, 4-54
  ERASEW, 4-55
  ERIN, 4-57
  EROF, 4-58
  EXIT, 4-59
  FIELD, 2-18, 2-31, 4-63
  FM, 4-71
  FOR, 4-72
  FORMAT, 4-75
  FRESH, 4-77
  GETFMT, 2-32, 4-79
  GOTO, 4-80

HOME, 4-81
IF, 4-82
INS, 4-84
INT, 2-15, 4-85
LBREAK, 4-86
LOG, 4-87
NL, 4-89
OPEN, 4-90
PAn, 4-92
PFn, 4-93
PRINT, 4-94
PROG, 2-3, 4-95
PROMPT, 2-49, 4-98
PUTENV, 4-100
READ, 4-101
RESET, 4-104
RETURN, 4-105
RUN, 4-106
SCRIPT, 2-3, 4-107
SERINIT, 2-47, 4-110
SHOW, 4-114
SWITCH, 4-116
SYSREQ, 4-118
TAB, 4-119
TEXT, 4-120
TIMEOUT, 4-122
TRACE, B-1
WAIT, 2-49, 4-124
WHILE, 4-127
WINDOW, 4-129
WRITE, 4-132
WTO, 4-134
COMMENT (/*) command, 4-38
CONNECT command, 4-39
Constants,
integer, 2-11
string, 2-11
Conventions,
commands, 4-3
data entry, 1-6
documentation, 1-5
functions, 4-137
COPY command, 4-43
CURSOR command, 4-45
CYCLE command, 4-46


# D


$DATE function, 4-144
$DATES function, 4-145
$DAY function, 4-146
Debugging facilities, B-1
Declaring variables, 2-13
Definitions, 1-7
DEL command, 4-47
DISCON command, 4-48
DUMP command, B-3
DUP command, 4-50


# E


EJECT comand, 4-51
ENDFMT command, 2-29
ENDP command, 2-3, 4-52
ENDS command, 2-3, 4-53
ENTER command, 4-54
Environment variables,
3270 Emulator+, F-2
D3274, F-2
Directory, F-3
Host/Local Session, F-3
Terminal, F-2
Terminal Information, F-3
ERASEW command, 4-55
ERIN comand, 4-57
EROF command, 4-58
Error messages,
execution, A-1
run time, A-12
syntax, A-1
$EVAL function, 4-147
EXIT command, 4-59
Expressions,
integer, 2-25
relational, 2-25
string, 2-26


# F


FIELD command, 2-18, 2-31, 4-63
Field variables, 2-18
$FLDADDR function, 4-150
FM command, 4-71
FOR command, 4-72
FORMAT command, 4-75
FRESH command, 4-77
Function summary table, 4-139
Functions,
$ATTR, 4-141
$CHDATE, 4-143
$DATE, 4-144
$DATES, 4-145
$DAY, 4-146
$EVAL, 4-147

$FLDADDR, 4-150
$GETCUR, 4-151
$GETENV, 4-152
$GETPID, 4-153
$GSUBSTR, 4-154
$HEX, 4-156
$ITOS, 4-157
$LENGTH, 4-158
$MONTH, 4-160
$NEXTFLD, 4-161
$RESP, 4-163
$SCAN, 4-165
$SEC2TIM, 4-168
$STOI, 4-169
$STRIP, 4-170
$TAB, 4-171
$TIM2SEC, 4-174
$TIMDIFF, 4-172
$TIME, 4-173
$YEAR, 4-175

## G

$GETCUR function, 4-151
$GETENV function, 4-152
GETFMT command, 2-32, 4-79
$GETPID function, 4-153
Global and local variables, 2-13
GOTO command, 4-80
$GSUBSTR function, 4-154

## H

$HEX function, 4-156
HOME command, 4-81

## I

IF command, 4-82
INS command, 4-84
INT command, 2-15, 4-85
Integer,
array variables, 2-15
constants, 2-11
expressions, 2-25
variables, 2-15
$ITOS function, 4-157

## K

Key sequences,
AT&T 4410 and
Teletype 5410 terminals, E-4
AT&T 4418 and
Teletype 5418 terminals, E-6
AT&T 4425 and
Teletype 5425 terminals, E-8
AT&T 605 Business Communications
Terminal, E-10
AT&T 610, 615, 620, and 630 Business
Communications Terminals, E-12
DEC VT100 terminal, E-14
standard ASCII terminals, E-2

## L

LBREAK command, 4-86
$LENGTH function, 4-158
Local screen format, 2-29
defining, 2-31
definition area, 2-29
loading, 2-32
multiple format files, 2-30
spilled files, 2-30
LOG command, 4-87
LSGEN,
accessing, 6-3
copy fields, 6-16, 6-30
create fields, 6-13, 6-29
cursor movement keys, 6-10, 6-32
define attributes, 6-23
define length, 6-22
define name, 6-21
delete characters, 6-11, 6-31
delete fields, 6-15, 6-30
delete lines, 6-12, 6-30
edit mode, 6-9
enter field definition mode, 6-20, 6-29
error messages, 6-27
exit field definition mode, 6-26
field definition mode, 6-19
field display, 6-17, 6-30, 6-31
insert characters, 6-11, 6-31
insert lines, 6-12, 6-31
move fields, 6-16
on-line help, 6-6, 6-29
operator information, 6-5, 6-31
quitting, 6-7, 6-29

## M

$MONTH function, 4-160
Multiple key sequences, defining, F-5

## N

Naming variables, 2-13
$NEXTFLD function, 4-161
NL command, 4-89

## O

OPEN command, 4-90
Operators,
  arithmetic, 2-22
  precedence of, 2-24
  relational, 2-22
  string concatenation, 2-23

## P

PAn command, 4-92
Parameter passing, 2-38
PFn command, 4-93
Pipe file,
  reading from, G-7
  writing to, G-8
Port initialization,
  asynchronous communication, 2-47
Precedence of operators, 2-24
PRINT command, 4-94
PROG command, 2-3, 4-95
Program,
  requirements, 2-3
  structure, 2-4
PROMPT command, 2-49, 4-98
PUTENV command, 4-100

## R

READ command, 4-101
Regression testing, G-4
Relational,
  expressions, 2-25
  operators, 2-22
Reserved words, 2-9
RESET command, 4-104
$RESP function, 4-163
RETURN command, 4-105
RUN command, 4-106

## S

Sample program,
  asynchronous host, 3-39
  synchronous host, 3-3
$SCAN function, 4-165
Screen buffers, 2-37
SCREEN variable, 2-35
SCRIPT command, 2-3, 4-107
$SEC2TIM function, 4-168
SERINIT command, 2-47, 4-110
SHOW command, 4-113
$STOI function, 4-169
String,
  array variables, 2-17
  concatenation operators, 2-23
  constants, 2-11
  expressions, 2-26
  variables, 2-16
$STRIP function, 4-170
SWITCH command, 4-115
Synchronous Response/No-Response mode transactions, 2-41
SYSAID variable, 2-35
SYSPRMT variable, 2-35
SYSREQ command, 4-117
SYSRET variable, 2-36
  with CAPTURE ON command, 4-26
  with CLOSE command, 4-35
  with CONNECT command, 4-39
  with DISCON command, 4-48
  with LOG command, 4-87
  with OPEN command, 4-90
  with PUTENV command, 4-100
  with READ command, 4-101
  with RUN command, 4-106
  with TIMEOUT command, 4-121
  with WAIT command, 4-123
  with WRITE command, 4-131

## T

TAB command, 4-118
$TAB function, 4-171
Terminal customization, F-4
TEXT command, 4-119
$TIM2SEC function, 4-174
$TIMDIFF function, 4-172
$TIME function, 4-173
TIMEOUT command, 4-121
TRACE command, B-1

## U

Utilities,
  Asynchronous Host
    Soft Function Keys, 5-27
  Generating Screen Field
    Variables, 5-17
  Get Fields, 5-25
  Upload and Download, 5-3

## V

Variables,
  declaring, 2-13
  field, 2-18
  global and local, 2-13
  integer, 2-15
  integer array, 2-15
  naming, 2-13
  string, 2-16
  string array, 2-17

## W

WAIT command, 2-49, 4-123
WHILE command, 4-126
WINDOW command, 4-128
WRITE command, 4-131
Writing a tutorial script, G-2
WTO command, 4-133

## Y

$YEAR function, 4-175