System Interface Module (SIM) Manual

Oracle Hospitality
Columbia, MD USA

Documentation

Oracle Hospitality product documentation is available on the Oracle Help Center at http://docs.oracle.com/en/industries/hospitality/.

Revision History

Edition	Month	Year	Software Version
1st	November	2007	1.0
2nd	May	2014	1.6 MR8
3rd	August	2014	1.6 MR9
4th	December	2015	1.7
5th	September	2016	1.7.1

Table of Contents

Appendix D - Key Types, Codes, and Names

Appendix E - sendsim

Appendix F - Windows DLL Access

Appendix G - SIM Events

Glossary

Preface

In This Chapter

This manual describes the System Interface Module (SIM) of Simphony and its proprietary Interface Script Language (ISL). This manual provides information needed to develop an interface that facilitates communications between Simphony and various third-party systems by learning how to write scripts in ISL.

Audience

- Programmers
- MIS Personnel
- Installers/Programmers

What should the reader already know?

- How to program high-level languages, such as BASIC or C/C++
- How to implement an interface
- How to program a Simphony database

Recognizing Abbreviations, Conventions, and Symbols

This section describes the abbreviations, conventions, and symbols that are used throughout this manual.

Abbreviations

Certain phrases, as listed below, are abbreviated to make reading easier.

This abbreviation...	refers to...
Interface	a software interface developed by a third party for the purpose of facilitating communications with Simphony
Operator	anyone operating an Oracle MICROS Workstation, including employees, cashiers, managers, servers
System	Simphony
Third-party System	any other system interfacing with the Simphony

Conventions

The typographic conventions explained below make following written instructions simpler.

This typographic convention...	described as...	which is used to denote...	is shown in the following example(s)...
. . .	three dots in a column	that part of a program or script has been intentionally omitted	startprint . . . endprint

This typographic convention...	described as...	which is used to denote...	is shown in the following example(s)...
|	the pipe symbol	a list of options where only one option may be selected [Note: A pipe is not allowed within an ISL command or function syntax.]	[GE | LE | GT | LT]
_ _H	an alphanumeric string with an "H" suffix	hexadecimal numbers	30H
boldface or **BoldFace**	words appearing in a bold type font	• commands or functions that are the focus of the discussion • items that must be entered exactly as they appear	**cleararray** kitchen_msg **DbResetTtls (UINT…)**
	boldface words with mixed uppercase and lowercase letters and no spaces	function and command names	**GetHex** **WindowClose**
ellipses...	three dots following a word or word series	that similar elements may follow	[,*prompt_expression...*]
[keys]	a keyname inside brackets	keys on a PC or workstation keyboard	[Enter] [Tab]

This typographic convention...	described as...	which is used to denote...	is shown in the following example(s)...
Placeholder **or** [*placeholder*]	italicized words italicized words between brackets	information the user must supply information the user must supply, but WITHOUT the brackets	*ResetType, expression, file_num* [*expression*]
"Prompts"	words or phrases in quotes	prompts, text strings, and chapter names	"Press Enter to confirm" "MM/DD/YY"
`text`	a non-proportional font	code, program output, and error messages	`clear`array kitchen_msg
under_score	an underlined space appearing between words	argument names made up of two or more connected words	ver_rsp kitchen_msg
UPPER CASE	words shown in all capital letters	modules, fieldnames, type definitions, system variables, messages, and certain key words	27FLEDEF.H @CHK_OPEN_TIME EMPL_DEF SECOND LAST_CHK_NUM

Symbols

Note: *This symbol is used to bring special attention to a related feature.*

Tip: *This symbol is used to indicate a special tip for using the current feature.*

Warning: *This symbol indicates that care should be exercised when programming a feature or performing an action.*

Chapter 1

Understanding the SIM and ISL

In This Chapter

This chapter contains an overview of the System Interface Module (SIM) and the Interface Script Language (ISL).

Getting to Know the SIM and ISL

The System Interface Module (SIM) extends the standard operation and functionality of Simphony through the Oracle Hospitality proprietary Interface Script Language (ISL). The SIM and ISL work together to provide establishments with the capability to enhance daily operations quickly and easily.

In this section, SIM and ISL will be introduced and how this module and script language work together will be explained.

What is the System Interface Module?

The SIM is the component of the Simphony allows the Simphony to interface to a variety of third-party systems. A special script language known as the ISL provides access to the SIM.

A property or enterprise will primarily use the SIM to communicate with third-party systems, such as a Property Management System (PMS) or pizza delivery system, by means of a special interface, called a SIM Interface. The user must develop a SIM Interface to facilitate the exchange of messages between the Simphony and the third-party system.

Note: *There will be properties and enterprises where SIM applications do not require communicating with a third-party system. In these cases, a SIM Interface is not required.*

What is a SIM Interface?

A SIM Interface is any software developed for the purpose of exchanging messages between a third-party system and the System Interface Module of Simphony.

For example, to develop a SIM Interface, use the same type of message formats and interface methods that would be employed to create a PMS Interface. Furthermore, SIM Interface must be enabled by setting up a link in the Simphony database, the same way PMS Interface would be enabled.

Accessing the SIM

A proprietary Oracle Hospitality interpreted language known as the ISL, facilitates access to the SIM. Through scripts composed with elements of the ISL, users can direct the SIM to execute a series of instructions, called events. SIM events are discussed later in this section.

Enabling the SIM

Although using the ISL is the method by which the user instructs the SIM to perform various functions, Simphony must be programmed to recognize and interpret the instructions contained in the script. Through database programming, a link between the script and the SIM Interface is formed, thereby, enabling the SIM. Once enabled, the SIM can carry out the instructions in the script if, at the workstation, an operator initiates an event by pressing a specially programmed key.

What is the Interface Script Language?

The ISL is a proprietary Oracle Hospitality interpreted language used to create small programs, called scripts. Contained in these scripts are the instructions that tell the SIM what to do.

The ISL includes easy-to-learn, easy-to-use language elements, including a repertory of commands, functions, and system variables, as well as simple statement formats. Users manipulate these language elements to create instructions that are executed when the script is run.

Users with programming experience and familiarity with script writing will quickly adapt to the ISL. And, although ISL is designed for use by systems developers, POS installers, and MIS staff, users with a strong knowledge of programming concepts and building blocks will also find the ISL easy to access.

Characteristics of the ISL

Like BASIC

ISL has been designed to closely resemble BASIC (or variants) in its structured, linear structure: The flow of scripts will be in a step-by-step structure but unlike BASIC, each line of the script does not need to be numbered. Like BASIC and other structured languages, the ISL supports decision-making language elements such as **If…Else**, and loop constructs using the **For…EndFor** and **Forever…EndFor**.

Language Elements and Components

A myriad of language elements, common to most interpreted languages, comprise the ISL to help build SIM applications. Among the language elements comprising the ISL are **commands**, **functions**, **system variables**, **operators**, and **format specifiers**. All of which will be used in script writing.

Numerous **commands** comprise the foundation of the ISL, allowing the designer to:

- Control the flow of instructions in the script
- Define and manipulate variable information
- Facilitate communications between a third-party system and Simphony
- Process input and output
- Handle a variety of file processing operations
- Send data to print devices

ISL **functions** enhance text handling and formatting facilities.

These additional elements are also provided:

- **System variables**, for reading selected definition and totals information from the database and setting certain system parameters
- **Operators**, relational and logical (Boolean), that perform mathematical actions on variable and constant operands
- **Format specifiers**, which when used with commands, allow the specification of the format of input and output data where permitted

For detailed descriptions about each component of a specific language element, refer to the chapters listed below.

Language Element	Where to Go
Commands	See “ISL Commands” on page 7-1
Functions	See “ISL Functions” on page 8-1
System Variables	See “ISL System Variables” on page 6-1
Operators	See “Using Variables” on page 4-1
Format Specifiers	See “Using Format Specifiers” on page 7-5

Event Procedures

ISL is also event-oriented. An **event** procedure is a group of statements and commands that is defined by the ISL **Event**…**EndEvent** commands. The scripts provide a frame-like structure for sequences of events.

In order to start an event, the event must be initiated with a specially programmed key or by a message response received from a third-party system. Once an event is initiated successfully, the SIM stops processing the script until another event is initiated.

For example, the event shown below performs the following set of tasks:

- Displays an ISL-defined window on the screen of the workstation,
- Prompts an operator to enter a room number, and
- Sends the room number and the number of the Transaction Employee to a third-party system (e.g., PMS).

```
event inq : 1                              // Execute when SIM Inquiry key 1
                                           //     is pressed
   var room_num : N5                       // Declare local variable
   window 2, 19, “Room Inquiry”            // Create input window
   display 2, 2, “Enter Room Number”       // Issue operator prompt
   input room_num, “ “                     // Accept input
   txmsg room_num, @tremp                  // Transmit room number and Transaction
                                           //     Employee to third-party system
   waitforrxmsg                            // Wait for response from third-party
endevent
```

- After the third-party system acknowledges receipt of the data, the event ends.

If the operator at the workstation were to initiate another event, the SIM would begin processing the script again but until then, the SIM waits for the next instruction.

Script Writing

What is a Script File?

The means by which the ISL issues instructions to the SIM is through small programs known as scripts. A script is an ASCII text file that can be created in any common text editor, such as Microsoft Notepad. These scripts can contain one or more events to implement SIM applications.

A single script must be maintained for each SIM Interface defined for a system. The script is linked to a SIM Interface through Simphony database programming. Once this relationship has been formed through database programming, the script can be executed by Simphony. For specific programming requirements, refer to "Programming Simphony for SIM" on page 2-10.

Being Familiar With Script Writing...

Script writing is a common way to issue instructions to a computer. However, scripts written with the proprietary Oracle Hospitality ISL have a specific format and include elements unique to this language. Consequently, Oracle Hospitality recommends that users familiarize themselves with the unique language elements and script structure before writing the first script. Refer to "Script Writing Basics" on page 1-1.

Being New to Script Writing...

This manual also contains a brief introduction to this method of automating operations. If the programmer has never used scripts before, Oracle Hospitality recommends reviewing this introductory material. For further details, refer to "Script Writing Basics" on page 1-1.

How the ISL Accesses the SIM

The SIM can be accessed through instructions executed by a small program called a script, written with the Interface Script Language (ISL). Within a script, there may be several events, each defined to perform different tasks.

Initiating an Event

Instructions within events in the script tell the SIM what tasks to perform. In order to carry out these instructions, the SIM first must be told to execute them. An event can be initiated within a script in one of the following three ways:

- The operator may press a **SIM Inquiry key**, programmed in the Simphony database to initiate an event.

- The operator may press a **SIM Tender key**, programmed in the Simphony database to initiate an event.

- A **third-party system**, interfaced to Simphony, can respond to a message sent to it by the SIM.

Pressing a SIM Inquiry or Tender Key

The flowchart on the next page illustrates what happens when an operator at a workstation presses a SIM Inquiry or SIM Tender key.

- First, the SIM verifies certain required parameters within the Simphony database.

- After verifying that certain programming options and links are set up, the SIM searches for the script.

- Once the SIM finds the correct script, it looks for a valid event, linked to the SIM Inquiry or SIM Tender key, that the operator pressed to initiate the whole process.

- Finally, after locating a valid event, the SIM runs the script and executes the instructions contained in the event.

Interfacing with a Third-party System

Several communications commands and system variables can be issued by events in the script. These commands and system variables can be used to send messages to a third-party system. In turn, the third-party system acknowledges these messages and responds over an interface method (i.e., TTY or TCP/IP) using a message format recognized by Simphony. Message formats and interface methods supported by the SIM are described in "Message Formats and Interface Methods" on page 2-3.

During this exchange, the SIM Interface acts as the go-between for both systems by shuttling the messages back and forth. A simplified version of the exchange goes like this:

- Simphony will transmit a message to a third-party system with a **TxMsg** command statement in a script.
- The SIM Interface will put the message in a format that is acceptable to the third-party system.
- Then, the third-party system will acknowledge the message and send back a response, such as data requested by the SIM.
- The SIM Interface will forward the response to Simphony in a message format acceptable to the POS: either fixed format or ISL format.

Features of the SIM

The SIM provides a variety of features to help create functional and useful SIM applications with the ISL. The main features of the SIM include support for:

- Common communications message formats and interface methods for development of SIM Interfaces
- A variety of methods of displaying, capturing, and printing information
- The standard communications protocol that is required to interface Simphony with a third-party system, such as a PMS or delivery system
- Comprehensive file I/O processing operations

Message Formats and Interface Methods

For developing a SIM Interface, the ISL handles two types of message formats: **fixed format** and **ISL format**. Both of these message formats can be sent over two different interface methods, including an **asynchronous serial interface** (Host TTY ports) and **TCP-based interface.**

A discussion of both the message formats and interface methods is in “Message Formats and Interface Methods” on page 2-3.

Methods of Displaying, Capturing, and Printing Data

Processing input and output, as well as printing data are the mainstay of most POS transactions. Consequently, the SIM handles a variety of input and output operations.

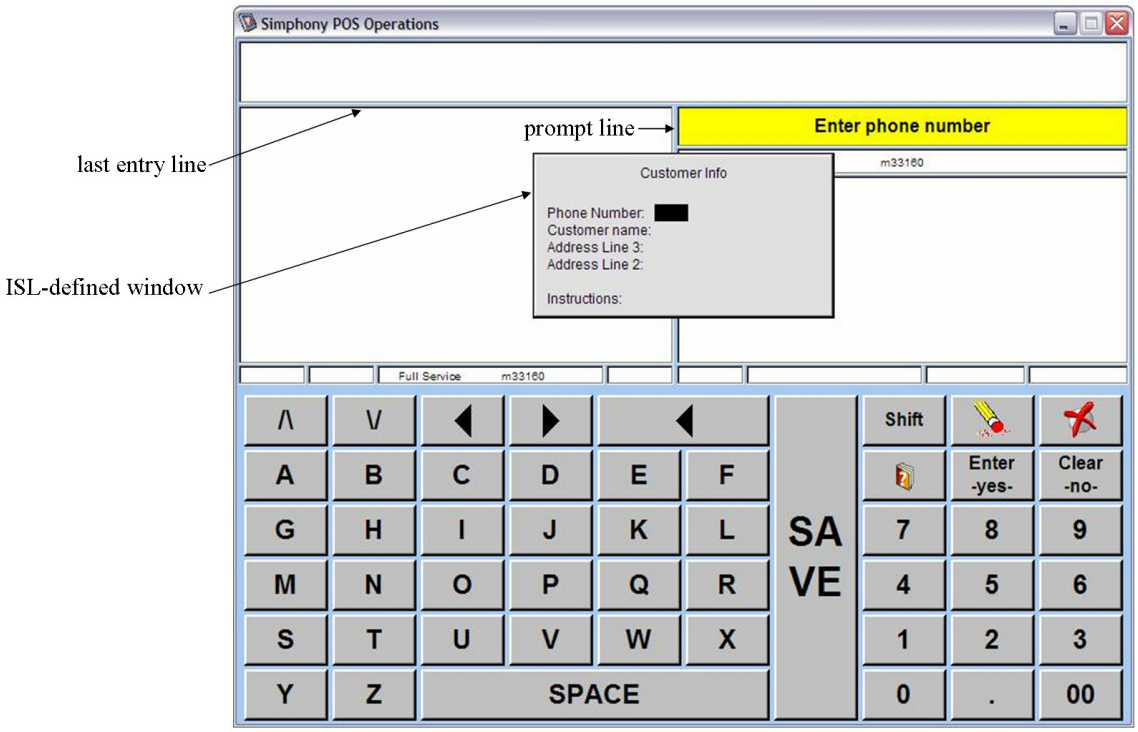
For displaying data, the Liquid Crystal Display (LCD) of a workstation is used as the platform for screen output.

Input data can be captured via the use of a touchscreen, a PC keyboard, a barcode reader, or a magnetic card reader.

The SIM controls printing with a versatile set of print commands and system variables, also called print directives.

The Liquid Crystal Display and Touchscreens

A script can be designed to display information in several places on the **Liquid Crystal Display** (LCD) of a workstation, including ISL-defined **windows**, the **prompt line**, and the **last entry line**.



Windows

Probably the most common place to both enter and display data is an **ISL-defined window**. Windows are drawn to the programmer's specifications and appear in front of the transaction detail and summary sections of the screen.

The Prompt Line

To help users step through operations more easily, user prompts can be provided on the **prompt line**. Also know as BOB, the Blue Option Bar.

The Last Entry Line

Error messages can be programmed to appear in the **last entry line**, to alert users of problems that occur while executing a script file.

Touchscreens

The SIM can make touchscreens pop up if they are defined by the *Touchscreen Style* module in the Simphony database within the Enterprise Management Console (EMC) or by the ISL Touchscreen Commands.

Programmed Touchscreens

Programmed touchscreens are already defined in the *Touchscreens* module of the Simphony database within the EMC. A programmed touchscreen that might pop up in an event is alpha or alphanumeric. For example, if an event prompts an operator to enter the name of a customer, the operator will need to enter the customer's name. The next statement in the event should be to display the workstation's default alpha touchscreen, so the operator can enter the customer's name.

ISL-Defined Touchscreens

To accommodate those instances where a programmed touchscreen is not appropriate, Oracle Hospitality provides several commands that let the ISL display touchscreens built within the script, or touchscreens created on-the-fly. An ISL-defined touchscreen comes in handy when function keys are needed for a very specific purpose and there is not a programmed touchscreen to accommodate the need.

For example, a user may be prompted to respond to a Yes-No question: The script instructs the SIM to display touchscreen keys [Yes] and [No]. Depending on which key the operator selects, additional instructions are carried out. For example, if the operator chooses [Yes], output from the transaction is sent to a KDS instead of the remote roll printer.

Data Entry

The entry of data is accepted from a PC keyboard, a touchscreen, a barcode reader, or a magnetic card reader. Normally, scripts that instruct an operator to collect data, such as name and address information, require entry using one of these methods.

Touchscreen

Data entry from a touchscreen is accepted, when required by the SIM during the execution of a script.

To support data entry from the touchscreen, the ISL includes commands for displaying a programmed or ISL-defined touchscreen when one is required. Normally, the script should display a touchscreen in order for the operator to select data from it. The data could be menu items, in which case, the script should pop up a sales transaction type touchscreen. Or, the script could direct the operator to enter the customer's name, in which case, an alpha touchscreen should pop up for the operator to use.

Magnetic Card Reader

Data that can be stored as track data on a magnetic card is also accessible by the SIM. Usually, track data includes the cardholder's name, a reference number, such as a credit card account number, and an expiration date. For instance, the account number of a country club member could be stored as track data, allowing the operator to capture it from the POS by swiping the member's card through the magnetic card reader.

Printing

Printing and backup printing are accomplished by the ISL through the use of several ISL commands and a variety of ISL system variables.

Print Commands and Print Directives

Print commands start print jobs, while the system variables, also called print directives, change the print characteristics of generated text. The **print directives** change the print type, similar to how standard parallel printer escape sequences work. For instance, if printing some text in red ink is desired, it can be accomplished with a print directive.

Type of Printers Supported

All output can be generated at roll printers and Kitchen Display Systems (KDS). For the WS4(+) only, the ISL accommodates printing to a laser or dot matrix printer that is connected to its parallel port and is configured for the Extended Line Printing option.

Interfacing with Third-party Systems

Communications are handled by the SIM through a variety of commands that support the exchange of messages between a third-party system and Simphony.

The type of messages sent back and forth between these two systems will typically include blocks of data. For instance, the third-party system may be used as a repository for data, such as customer name and address information in a customer database. This data might be used by Simphony to verify information in the third-party database with information input from the POS side.

In order for Simphony to send and accept data from a third-party system, a SIM Interface must be developed and enabled using the interface methods supported by the SIM, and the messages must be put in a format accepted by the SIM.

ISL File Handling

File processing operations found in other programming languages are also supported by the ISL. Files can be opened, and while open, read and write operations can be performed, then close the file. For example, an application at a country club may involve checking a file for a member account number, retrieving it, and adding it to a guest check. But, if the member's account number is not found in the file, this feature allows the ISL to assign an account number for the new member, with the next available account number in the file.

Creating SIM Applications with the ISL

The ISL is the gateway to the System Interface Module (SIM) of the Simphony software. When the ISL is used correctly, this powerful script language can be harnessed to build useful and practical SIM applications for all types of POS environments, including restaurants, bars, hotels, country clubs, etc.

Benefits of SIM Applications

The impact of SIM applications on these POS environments will be immediate, especially when used to make certain POS transactions and functions easier for operators to perform. Such applications have the added benefit of making Simphony easier and simpler to use, thus, improving overall customer service.

Equally important, is the fact that creating SIM applications also enable the user to expand the capabilities of Simphony. With these applications, Simphony can be taken beyond its traditional function and improve existing features. For example, guest charge posting could be made faster, user prompting and System messaging could be improved, and customers can be tracked through the collection of information, like names and addresses. Whether simple or complex, when implemented properly, these types of SIM applications can make the system a more powerful tool for users.

Types of SIM Applications

Although SIM applications can emphasize a variety of features, the applications likely to implement might involve such features as:

- Collecting and saving data for future retrieval or tracking, such as tracking customer sales in order to generate coupons, as rewards or incentives for frequent diners
- Expediting certain POS operations, like automatically applying a discount to a guest check when certain conditions are met
- Communicating with a third-party system, such as a Pizza Delivery System
- Writing to and reading from files, as in the case of verifying input with information in a file stored on disk
- Printing and posting information to checks in ways that the system may not be programmed to do so, like generating messages with special instructions for kitchen staff

In this section, there are several real-world examples of SIM applications, implementing a combination of SIM features. These examples are provided to give an idea of the type of SIM applications that can be created with the ISL. Keep in mind: These are only examples, and implementations of similar SIM applications can vary.

Generating Coupons for Customers

The collection of data, such as customer information, is easily handled by the SIM. Such data can be captured and used by a third-party delivery system, interfaced with Simphony, allowing the entry and recall of customer name and address information.

Since customer sales and other data can be tracked this way, the ISL makes it easy to develop SIM applications that create rewards for customers. Most common are rewards in the form of coupons and other incentives. Customers redeem these coupons for cash, gifts, reduced purchase price, etc.

Frequent Diners

Customers who frequently patronize an establishment are often targeted to receive rewards and incentives for their continued loyalty. At the fictional restaurant, a Birthday Club is one of the incentive programs offered to frequent diners.

Assume that a third-party database exists with pertinent information about frequent diners, such as name, account number, birthdate, etc., for sales tracking purposes. In addition, also assume that the restaurant issues frequent diners a special VIP card, including an account number and birthdate as track data[1]. Whenever the frequent diner visits the restaurant, the diner must present their VIP card so the server can credit sales to the frequent diner's account.

1. This term refers to information stored on the magnetic stripe of a credit card. The information is often stored on separate areas of the magnetic stripe, called tracks. For example, the account number and birthdate in this example are stored on Track 2 of the magnetic stripe.

Generating a Birthday Coupon

A server generates a "Birthday Coupon" based on the information that the restaurant is tracking. The coupon is redeemable for a free appetizer upon the customer's next visit. At this restaurant, the Birthday Club works like this:

- Using the birthdate read by a magnetic card reader from the VIP card track *data*[1], Simphony checks it against the current system date. If the birthdate is equal to the system date, then Simphony will automatically issue a Birthday Coupon when the server closes the guest check.

- When presenting the guest check, the server also gives the diner the "Birthday Coupon."

Interfacing with a Pizza Delivery System

One of the important features of the ISL is its ability to support an interface between Simphony and a third-party system. Such an interface allows both systems to communicate with each other by handling the exchange of messages between the systems.

For example over a SIM Interface, Simphony captures customer information input on the POS side, then the information is sent to the third-party delivery system. A SIM Interface enables the operator to retrieve this information from the delivery system's database, as needed, through a query or search.

Guest Check Information Detail

Information that might be captured and stored in a third-party database includes the name and address information, the customer's telephone number, as well as directions to the customer's residence. When a customer telephones the pizzeria, an operator uses the customer's telephone number to access any existing information in the pizza delivery system's database.

As an added benefit, this SIM application also prints this customer information on the guest check. For instance, the directions to the customer's residence prints on the check for the delivery person to reference.

1. This term refers to information stored on the magnetic stripe of a credit card. The information is often stored on separate areas of the magnetic stripe, called tracks. For example, the account number and birthdate in this example are stored on Track 2 of the magnetic stripe.

Previous Order History

In addition to name and address information, a customer's order history is also information that the third-party system maintains. By retrieving previous order history, the operator eliminates the additional steps required to post order information, namely menu items, to a guest check. If the customer always orders a large pepperoni and mushroom pizza, these items can be retrieved from the third-party database, and posted to the check, without directly accessing the menu item keys again. In a fast-paced environment like a pizza delivery operation, where quick turnaround is critical, eliminating steps in the order entry process is a real advantage.

Collecting Customer Information for a Membership List

Many implementations of SIM applications collect data, such as name and address information or guest history. Collecting and storing this information can be retrieved for later use.

The ISL can be used to handle these functions in one of two ways, including:

- Use ISL file I/O capabilities to read from and write to files that are stored on the same PC running Simphony.
- Interface with a third-party system, as described in the pizza delivery system example on page 16, to send data to and retrieve it from another system.

Although both options enable the storage and retrieval data, the first option allows faster access data, since no communications overhead is involved.

Reading Data From a File

One implementation of the file I/O option is to collect customer data and add it to a membership list, which is kept in a file. The application requires that an operator at a country club, for example, enters name and address information in Simphony. Behind the scenes, the ISL opens the file and writes this new information to it, or verifies existing information, then closes it.

The country club uses this file to verify membership information. For instance, by reading this file, operators determine whether the customer's membership dues are current, or whether the member is entitled to certain privileges, such as running a tab at the country club's bar.

Customizing Output

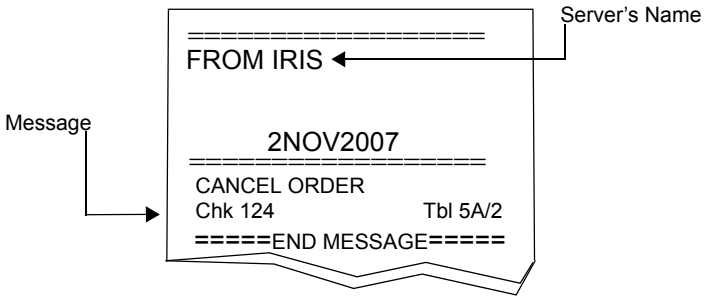
Printing is an essential POS function that can be expanded to accommodate the unique preferences of a property. Simphony comes equipped with many different programming options that affect how things are printed, where they print, and when they will print. With the ISL, printing can be further customized for a property by extending the functionality of Simphony beyond its normal programmable limits.

The ISL's special printing abilities in the previous examples, such as printing customer information on guest checks and generating coupons for customers has been previously mentioned. Now let's examine how servers' can save time when they must send special instructions to kitchen staff about orders.

Sending Special Messages to the Kitchen

The SIM application allows an operator to pick a message from a list, then send the message to the kitchen. The pick list includes messages frequently sent to the kitchen, which is a feature that current programming options for number lookups (NLUs) do not support.

A remote roll printer in the kitchen generates the following message on a chit:



Creating Raffle Tickets

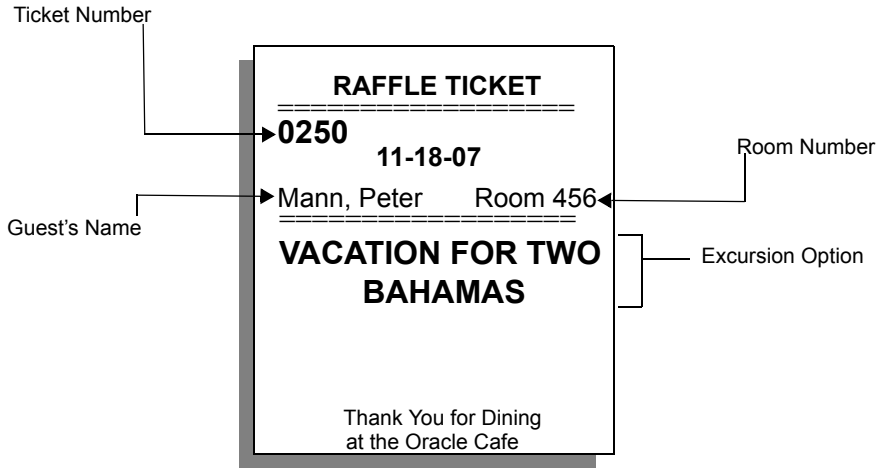
Another good example of how the special printing capabilities of the ISL can be used, is creating output other than guest checks, remote order chits, customer receipts, etc.

Earlier in this section, how to generate a coupon with the ISL was described. Now, let's create a raffle ticket for hotel guests.

This application requires that the Simphony interface with a PMS. The application performs a common PMS task: collecting a guest's name and room number when the operator closes a guest check in one of the hotel's food and beverage outlets.

For this application, the ISL verifies the input with data already stored in the PMS database. Based on how many times the guest has patronized the hotel's four star restaurant, for example, a raffle ticket chit is generated after the guest check prints.

The raffle ticket includes an excursion option, which was selected from a pick list of others, as well as the guest's name and room number. The ISL also automatically assigns a ticket number to the chit.



Such a chit cannot be generated by Simphony using current programming options. However, the ISL makes it possible to print output that is outside the traditional realm of a POS.

Chapter 2

Getting Started

In This Chapter

This chapter contains a description of the message formats and interface methods that must be used to develop a SIM Interface. Also included, is a description of the Simphony database programming required to enable the SIM and a SIM Interface.

Getting Started with the ISL and SIM

Understanding exactly what purpose the System Interface Module (SIM) and the Interface Script Language (ISL) serve, as well as how they interact, are the first steps that should be taken toward learning how to create SIM applications. Chapter One answers these questions about the SIM and the ISL.

This chapter focuses on the steps that must be taken in order to implement the SIM applications successfully. In this chapter, the following is described:

- What communications message formats and interface methods are available for developing a SIM Interface
- How to enable the SIM and a SIM Interface

Developing the SIM Interface

If creating SIM applications that require communicating with a third-party system, such as a PMS or delivery system, a SIM Interface must be developed. This interface must utilize supported message formats and interface methods to facilitate communications between Simphony and the third-party system. In “Message Formats and Interface Methods” on page 2-3, the message formats and interface methods to apply when creating the SIM Interface is discussed.

Enabling SIM and a SIM Interface

There is specific Simphony database programming that must be defined to enable the SIM and SIM Interface and execute a script successfully. In this chapter, these requirements are summarized in “Programming Simphony for SIM” on page 2-10.

Message Formats and Interface Methods

To develop the SIM Interface, use the message formats and the interface methods described in this section.

Message Formats

There are two classes of message format: fixed format and ISL format.

Note: *Both formats over the same type of interface can be used. For example, the ISL format can be used over both a TTY- and a TCP-based SIM Interface.*

Fixed Message Format

Support for this message format allows a PMS compatible fixed-format messages to communicate with Simphony via a SIM Interface. However, Oracle Hospitality recommends that the ISL message format be used unless fixed-format messages are a requirement of the third-party system.

This recommendation is made since the fixed content of this message format provides a limited set of information. For instance, a fixed message must always contain four sales itemizers, four tax itemizers, etc. If a PMS needs access to eight sale itemizers, for example, then this message format cannot be changed to accommodate eight sales itemizers. The ISL message format provides much more flexibility.

Consequently, the fixed message format should only be used in cases where the third-party system requires fixed-format style compatibility due to pre-existing PMS installations.

Message Format

The format of the message is as follows:

SOH *ID* STX *Data* ETX *Checksum* EOT

ISL Message Format

Messages defined using the ISL are structured like fixed messages, except that the *Data* segment of a fixed message is broken out into two segments in the ISL message: *Application_Sequence* and *Application_Data*.

However, the segments of the ISL message are enveloped using control characters. IBM PC character codes, an ASCII superset, provide support for international characters.

The format of the ISL message is as follows:

SOH *ID* STX FS *Application_Sequence Application_Data* ETX *Checksum* EOT

A description of each message segment follows.

SOH

The SOH character (start of header) serves as a message lead-in character that identifies the start of a new message. The SOH character is represented by the 7-bit hexadecimal value 01H, plus a parity bit, if applicable.

ID

This is the POS Source ID segment that includes information about the workstation. The workstation initiates the message and identifies the interface.

The format of this segment is as follows:

Field	Length	Format	Remarks
POS Workstation Number	9 Bytes	2 or 9 ASCII Digits	May contain leading spaces or zeroes and is between 1 and 999999999, depending on 2/9 digit ID in the Interface module. Standard interface format is 2 digits.
Interface Name	16 Bytes	16 ASCII Characters	Uses IBM PC character set.

STX

The STX character (start of text) serves as a data field lead-in character that identifies the start of the message data block. The STX character is represented by the 7-bit hexadecimal value 02H, plus a parity bit, if applicable.

FS

The FS character (field separator) identifies this message as a SIM message data block. The FS character is represented by the 7-bit hexadecimal value 1CH, plus a parity bit, if applicable.

Application_Sequence

The *Application_Sequence* segment comprises a two-digit sequence number and a retransmission flag. Each POS workstation application increments its own sequence number with each message. When a message is being retransmitted, the same sequence number will be used as the original message. In addition, a retransmit flag character is provided.

The format of this segment is as follows:

Field	Length	Format	Remarks
Applications Sequence Number	2 Bytes	2 ASCII Digits	May contain leading spaces or zeroes and is between 00 and 99.
Retransmission Flag Character	1 Byte	Space or 'R'	'R' character (ASCII 52H) is placed in this field if this is a retransmitted message.

The *Application_Sequence* number is initially set to "01" when the application starts. The application rolls the sequence number back to "01" after "99."

If the third-party system receives a message containing the same sequence number as the previous message and the retransmit flag is set, the third-party should retransmit the last response.

Application_Data

Both the **TxMsg** and **RxMsg** commands define the *Application_Data* segment. The total size of the message can be 32K from the SOH to the EOT. There are a maximum of 37 bytes overhead, which means that the maximum byte count of all the fields and field separators is 32768 - 37, or 32731 bytes.

Note: *When the asynchronous serial interface messages are not limited, it is recommended that they be no more than 1024 bytes in length, so that the interface remains responsive.*

Multiple fields can comprise this segment. Individual fields within the *Application_Data* segment are separated by the ASCII field separator character (1CH), inserted by the ISL.

In addition, the first field within the receive message *Application_Data* segment defines the name of the ISL procedure to execute when processing the response message. The detail description of the **RxMsg** command describes the relationship between the command and the *Application_Data* segment of an ISL message.

ETX

The ETX character (end of text) serves as a data field lead-out character that identifies the end of the message data block. The ETX character is represented by the 7-bit hexadecimal value 03H, plus a parity bit, if applicable.

Checksum

The *Checksum* field is only used when communicating over an asynchronous serial interface. A TCP-based SIM Interface will ignore this field, so it can be omitted from the message format. When the *Checksum* is part of the message, however, format it as follows:

Field	Length	Format	Remarks
Checksum	4 Bytes	4 ASCII Hex characters	Contains ASCII characters in the range 30H - 39H and 41H - 46H (0-9 and A-F).

The *Checksum* is the 16-bit binary addition (excluding parity, if applicable) of all characters after the SOH, up to and including the ETX character. The *Checksum* is initially set to zero. For transmission, the *Checksum* is represented as four ASCII-Hex characters.

EOT

The End of Transmission character (EOT) identifies the end of the message. It is represented by the hexadecimal value 04H, plus a parity bit, if applicable.

Interface Methods

The SIM supports two types of interface: an asynchronous serial interface and a TCP-based interface. Both interface methods support the Simphony-compatibility fixed-format message and the ISL message format.

Note: *Both the fixed and ISL formats may be used over the same interface and, since Simphony supports multiple interfaces per Revenue Center, any combination of these interfaces may be used within a Revenue Center.*

Asynchronous Serial Interface

This type of interface method supports communications with a third-party system over an Asynchronous Communications Adaptor (COM port) installed in the PC that controls Simphony.

This interface is widely used to implement a PMS Interface to facilitate communications between Simphony and a PMS. With the introduction of SIM, this interface method can also be a choice for implementing a SIM Interface.

Asynchronous Serial Interface Specifications

Specifications for developing an asynchronous serial interface are defined in the *1700/2000/4700/8700 PMS Interface Specifications Manual*. Refer to this manual for more information about this type of interface method.

Configuring a TTY Interface in the Simphony Database

If using this type of interface, the Enterprise Management Console (EMC) *Hardware* | *Interface* module must be configured in order to enable the interface.

To enable a TTY interface:

1. Add an interface record.
2. Select **TTY** in the **Comm Type** field.
3. Enter the device name in the **Comms Name** field. For example, type:

- `tty2a` (which represents the TTY device), or
- `workstation` (which indicates that the workstation is directly connected to the PMS)

4. Choose the appropriate number of digits for the interface by enabling or disabling this option:
   ON = Use 9 digits for Terminal IDs; OFF = Use 2 Digits for Terminal IDs

TCP Interface

This interface is designed to connect Simphony to Windows®-based systems and other systems using the TCP/IP networking protocol. This interface can also be used to facilitate communications between the POS application and third-party applications that reside on the same Windows platform as the Simphony software.

This interface is also compatible with many forms of local area networks (LANs), including Ethernet, Token Ring, FDDI, Arcnet, PPP, etc.

Configuring a TCP Host in Simphony

If using this type of interface, the Enterprise Management Console (EMC) *Hardware* | *Interface* module must be configured in order to enable the interface.

To enable a TCP interface:

1. Add an interface record.
2. Select **TCP** in the **Comm Type** field.
3. Enter the **TCP Host Name**.
4. Choose the appropriate number of digits for the interface by enabling or disabling this option:
   ON = Use 9 digits for Terminal IDs; OFF = Use 2 Digits for Terminal IDs

TCP Connection

The SIM connects to the TCP port as a client. The SIM Server should accept TCP connections from the Simphony POS client on the port "micros-sim." If this service is not defined, the port number 5009 should be used as the default.

Error-Handling

If the receiving system detects an error in the message or some other applications-related error, it should provide an appropriate error message response to the POS application. In addition, if using the fixed message format, response messages should handle error messages. As such, to support these conditions, the ISL should define specific error responses.

Pinging

The TCP connection has a typical "keep-alive" time-out of two hours. In order to detect a "down" interface more quickly and re-establish the connection, the SIM periodically sends a "ping" message to the server about every five minutes. The server should detect the ping message and return the message in its original format.

The format of the ping message should be as follows:

SOH *ID* STX ETX EOT

The *ID* source segment contains a null address: the POS wokrstation number will be zero and the interface name will contain spaces.

TCP Interface Code Example

Several programming aides are provided in “TCP Interface Code” on page B-1. Two samples of code are provided to implement a TCP Interface: a SIM TCP Server and a SIM Server.

Programming Simphony for SIM

After creating a SIM Interface, the Simphony database must be programmed to

- Enable the System Interface Module (SIM)
- Activate the SIM Interface through an Interface link
- Link the scripts to the appropriate SIM Interface

To perform the tasks described above, complete the programming action steps in this section. This section includes a quick reference programming chart to help experienced Simphony programmers get started quickly. More in-depth descriptions of the programming action steps are also in this section.

Prerequisites

In order to effectively implement the programming action steps described in this section, the programmer should have an understanding of:

- Simphony database structure
- POS database programming concepts

Database Programming Quick Reference

For quick reference, the table below outlines the action steps that must be completed in order to program the database. Each of these action steps is discussed briefly on pages 2-12 through 2-14.

Action Steps		EMC Modules/ Programs Affected	Procedure
1	Define Interface	Interface module	Define the interfaces.
2	Name and Store Script	None	Assign the filename pms###.isl to script file, where ### is the object number assigned to the PMS. Store pms###.isl in *\Micros\Simphony\etc*
3	Link the ISL Script File to an Interface	RVC Parameters module	Define PMS for RVC; use object number to name the script. PMS object number pms**2**.isl
4	Create SIM Inquiry Key	Touchscreen Design module	Define key; see table on page 2-13.
5	Create SIM Tender Key	Tender Media module Touchscreen Design module	Create new tender key or use existing tender key, and • Enable PMS Option **Use ISL TMED Procedure Instead of PMS Interface**. Assign Interface to key.

Database Programming Action Steps

This section discusses these action steps in a summary fashion.

Step 1: Define the Interface

EMC Module: *Interface* module

Add a record for each interface connected to Simphony. Make a note of the interface object number, this object number will be used to define the script filename in Step 2.

Step 2: Name and Store Scripts

- **Name** the script **pms###.isl**, where "###" is the Interface object number in the Simphony Interface module. For example, the file for Interface object number 4 must be called **pms4.isl**. There should be no leading zeros in the numeric portion of the filename.

  -OR-

  Give the script **any name,** up to eight characters long, and then add the **.isl** extension. Remember the .isl extension does not need to be added when configuring this name in the *Interface* module's **ISL Script Name** field. The system assumes the extension.

- **Store** the script in the *\Micros\Simphony\etc* directory. If the filename is defined in the Interface table, the SIM will first look for that file; if the filename is blank, then the SIM will look for **pms###.isl**. If the SIM cannot find pms###.isl in that directory when an event is initiated, the SIM will look for the default script *\Micros\Simphony\etc\script.isl* instead. In the instance when pms###.isl or script.isl cannot be found, the SIM will issue an error.

Step 3: Link Scripts to a SIM Interface

Database Module: *RVC Parameters* module

Enable an Interface link that corresponds to the Interface object number assigned to the script in Step 4. Each Revenue Center supports multiple SIM Interfaces, all of which can be defined in the *Interface* module through an interface link.

Step 4: Create a SIM Inquiry Key

Database Module: *Touchscreens* module

Create a SIM Inquiry key to initiate a corresponding event in a script.

Pressing a SIM Inquiry key is one of the three ways to initiate an ISL event. The key is linked to the event with a function key code that tells the SIM two things:

- Where to look for the event and
- Which event to execute.

The SIM key codes are programmed in the *Touchscreen Design* module. There are 20 valid keys for each of the PMS Computer links available in a Revenue Center.

For example, assuming that an Event Inq: 4 exists in the script pms1.isl, one can determine from the table below that the event should be linked to SIM 1 Inquiry Key 4, and assigned to function key code 924.

SIM Inquiry Keys	Corre-sponding Key Codes	RVC/ PMS Record #	Sample PMS Object Number and Name	Sample File Name
SIM 1 Inq 1 - 20	920 - 939	1	#1 / Fidelio System	pms1.isl
SIM 2 Inq 1 - 20	940 - 959	2	#2 / Property Mgmt System X	pms2.isl
SIM 3 Inq 1 - 20	960 -979	3	#14 / Property Mgmt System Y	pms14.isl
SIM 4 Inq 1 - 20	980 - 999	4	No PMS Link	none
SIM 5 Inq 1 - 20	1000 - 1019	5	No PMS Link	none
SIM 6 Inq 1 - 20	1020 - 1039	6	No PMS Link	none
SIM 7 Inq 1 - 20	1040 - 1059	7	No PMS Link	none
SIM 8 Inq 1 - 20	1060 - 1079	8	No PMS Link	none

Step 5: Create a SIM Tender Key

Database Modules: *Tender Media* module
Touchscreen Design module

Create a Tender key to initiate a corresponding ISL tender event in a script.

Another way to initiate an event is by pressing a SIM Tender key. Like SIM Inquiry keys, SIM Tender keys must be linked to a corresponding event procedure. In the case of the a SIM Tender key, the tender/media event is the only valid event initiated by this key.

Note: *The tender event must be executed while in a transaction, i.e., while a guest check is open.*

To program a SIM Tender:

- Select a Tender key to act as a SIM Tender. If necessary, create a new tender in the *Tender Media* module.
- Link the key to a SIM Interface, defined by its PMS Computer link, by
    - Enabling the PMS Option, **Use ISL TMED Procedure Instead of PMS Interface** in the *Tender Media* module.
    - Assigning a PMS Record Number to the key in the *Tender Media* module.
- Link the tender event to the SIM Tender key by using the object number assigned to it in the *Tender Media* module. For example, if the Cash key is defined as object number 10 in the *Tender Media* module, then use this object number in the **Event Tmed** command syntax (i.e., Event Tmed : 10).

Chapter 3

Script Writing Basics

In This Chapter

For those users who are new to script writing or need to familiarize themselves with script writing conventions, this chapter discusses some basic script writing concepts to apply when creating scripts.

Getting Started with Script Writing

This section introduces the process of script writing, a tool used to create SIM scripts.

This section provides the specific conventions and formats needed in order to write scripts with the ISL. This section also includes the information needed to begin using scripts for the first time.

Specifically, this chapter covers:

- What a script is
- Why use scripts
- What the parts of a script are
- How to write a script
- What is proper script writing style
- How to test a script
- How to document the script for others
- How to encrypt the scripts

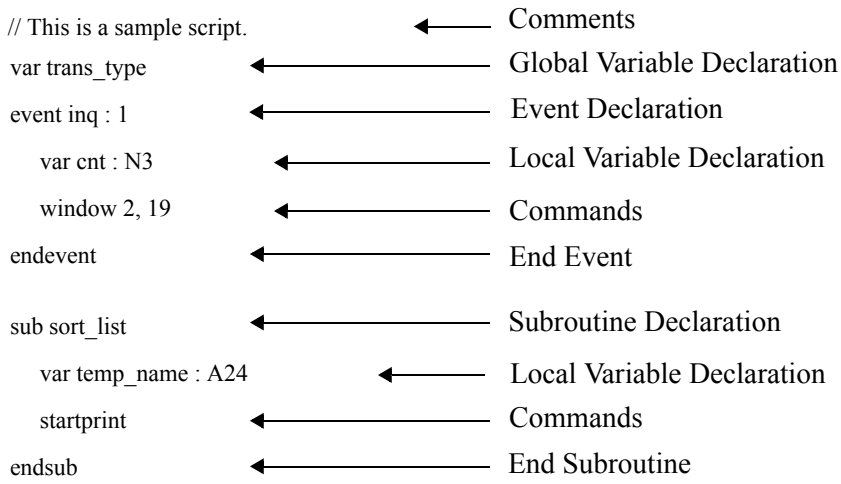
What is a Script?

The means by which the ISL issues instructions to the SIM is through small programs known as scripts. A script is an ASCII text file that the programmer creates in any common text editor, such as Windows® Notepad. These scripts can contain one or more events to implement SIM applications.

A separate script must be maintained for each SIM Interface defined for a system. The script is linked to a SIM Interface through Simphony database programming. Once this relationship is formed through database programming, the script can be executed by the SIM. For specific programming requirements, refer to "Programming Simphony for SIM" on page 2-10.

Structure of a Script

The basic structure of scripts should be written in a format similar to the sample script below. When writing the script, keep the following structure in mind. A brief description of each part follows the diagram.



Comments—Comments are used to document the purpose and scope of events included in the script. Place comments anywhere in the script, but make sure each comment line is preceded by two backslash characters "//" (see page 3-9).

Global variable declarations—Global variables are initialized at the beginning of each script and maintained for the duration of the script.

Event declaration—ISL is event-oriented. Almost all ISL statements will be contained in events or subroutines, called by those events (see page 7-58). There are five types of events:

- **Event Inq—**This event executes when a SIM Inquiry key is used.
- **Event Tmed—**This event is executed when a SIM Tender key is used.
- **Event RxMsg**—This event is executed when a response is received from the third-party system.
- **Event Final_Tender—**This event is executed whenever the last tender event has occurred, but before a check has closed.
- **Event Print_Header and Print_Trailer—**This event is executed whenever certain control characters are programmed in the *RVC Descriptors* module within the Enterprise Management Console (EMC).

Local variable declaration commands—Local variables are purged after each event completes execution; the event terminates after an **EndEvent**, **ExitCancel**, **ExitContinue**, or any other command that causes the event to stop executing, successfully or not.

Subroutine declaration—Subroutine procedures are called from other event or subroutine procedures, allowing common code to be used by multiple events. As a script writing convention, Oracle Hospitality recommends that all subroutines be placed after all events in the script. Subroutines are described on page 7-166.

Creating Scripts

Scripts consist of one or more events. Each event within the script represents a task that the SIM should perform. For instance, Chapter 1 discusses several different types of SIM applications. Each of these applications can exist as separate events within the same script.

Guidelines for Creating Scripts

When beginning to develop the SIM applications, use the guidelines below to create the script.

- Understand the tasks that the script will perform.
    - Outline the structure of the script. In plain English, write down the steps needed to automate in each event. Carefully, note each detail about the tasks that the event will accomplish.
    - Write a brief description of the task each event or subroutine performs. For future reference, this description should appear as a comment before each event and subroutine in the script.
    - Note any input and output. Does the user enter data? Does a guest check, remote order chit, etc. need to be generated?
    - Think about what assumptions being made about the environment. For instance, if the application applies a discount to a check total greater than or equal to $50.00, test for this case in the script before applying the discount.
    - Protect against the user. The event should not allow users to get stuck or to perform a task that they should not. For example, if input is requested from an operator and the operator gives incorrect input, then an error should be issued to force the operator to enter the requested information.
- Determine what variables (global, local) need to be declared. For a discussion of user variables, refer to “Using Variables” on page 4-1.
- Determine whether some of the events in the script perform similar tasks. If they do, consider creating a subroutine to save time in writing the script and to make the script process instructions more efficiently.
- Translate the instructions written into one or more lines of code.
    - Consult the ISL Quick Reference beginning on page 8-1 to determine

what commands, functions, and system variables allow the desired task to be performed.

- Learn what considerations are involved in writing ISL statements. “Script Writing Style” on page 3-9 covers these guidelines.

- Review the detail descriptions of the required language elements. The examples provided with detail descriptions can be helpful as templates from which ISL statements can be built.

Examples of Scripts

Charge Denial

This script places a window with the title "Charge Denied" on the screen, and provides text in it with the reason why the charge was denied. The operator is then prompted to enter the [Clear] key before cancelling the operation.

```
event rxmsg : denial_msg
    var reason_text : a32

    rxmsg reason_text
    window 3, 34, "Charge Denied"
    display 2,@CENTER, reason_text
    waitforclear
    exitcancel
endevent
```

Charge Posting

This script posts a charge to a PMS system and waits for the response.

```
var guest_id:a20

event tmed : 1                          //PMS room charge key is
                                        // Tender key #1
    input guest_id, "Guest posting, enter name or room #"
    txmsg "CHG_POSTING", guest_id, @TNDTTL, @CHKNUM,@RVC
    waitforrxmsg
endevent

event rxmsg : chg_posting
    var status : a1, message : a20, temp_tndttl : $12

    rxmsg status, message, temp_tndttl

    if status="P"                       //Posting approved
        saverefinfo message
        @TNDTTL = temp_tndttl
        exitcontinue
    endif

    if status = "D"                     //Charge declined.
        exitcancel message
    endif


    if status = "E"                     //Error
        exitwitherror message
    endif
endevent
```

Address and Phone Number Entry

This example prompts the operator to enter a customer's address and phone number.

```
var phone_num : N7              //Global variable, keep phone
                                // number around.

event inq : 1                   //Use Inquiry key #1 to get guest info.
    var cust_name : A32
    var addr1 : A32
    var addr2 : A32
    var addr3 : A32
    var special_instructions : A32

    window 7, 50, "Customer Info"
    display 2,2, "Phone Number:"
    displayinput 2, 18, phone_num{:###-####}, "Enter phone number"
    display 3,2, "Customer name:"
    displayinput 3,18,cust_name, "Enter customer's name"
    display 4,2, "Address Line 1:"
    displayinput  4,18,addr1, "Enter address"
    display 5,2, "Address Line 2:"
    displayinput  5,18,addr2, "Enter address"
    display 4,2, "Address Line 3:"
    displayinput  6,18,addr3, "Enter address"
    display 7,2, "Instructions:"
    displayinput 7,18,special_instructions, "Enter instructions"
    windowinput

    txmsg phone_num, cust_name, addr1, addr2, addr3, \
        special_instructions
    waitforrxMsg
endevent

event rxmsg : phone_num
    var phone_num : n10
    var status : a1

    rxmsg status, phone_num
        if status = "N"
            exitwitherror "Phone Number Not Found "
        endif
    window 3, 12
        display 2, @CENTER, phone_num
    waitforenter
endevent
```

Script Writing Style

Before beginning to write the first script, review the style conventions in this section. For readability, apply these conventions in the script writing.

Case

ISL statements are not case sensitive and the use of case in examples is purely for clarity and the author's choice of style. However, quoted strings are case sensitive.

Length of Variables

The maximum character length for all variable names (user variables, subroutines, etc.) is 255.

Comments (//)

Declarations and commands will always be on their own line and should be the first non-white space characters. A comment may be placed on a line by beginning the comment with "//" characters. All characters to the right of the comment identifier "//" are ignored. For example:

```
Window 2,19, "ROOM INQUIRY"          //Create window
Display 2,2,  "Enter Room Number"    //Prompt for room number
```

A comment may reside on its own line or to the right of a command and its arguments. Lines should be terminated with an ASCII carriage return, or an ASCII linefeed, or a carriage return/line feed pair.

Continuation Lines (\)

A line continuation character "\" is provided to allow commands to continue from one line to the next. For example, a command that overflows several lines might be:

```
TxMsg field1, field2, field3, field4, \
field5, field6, field7, field8, field9
```

It is not possible to break apart a string with a line continuation character:

Correct:

```
errormessage "Choose a number between 1 ",\
             "and 10"
```

Incorrect:

```
errormessage "Choose a number between 1\
             and 10"
```

Line continuation characters may ***not*** be followed by comments or any other commands.

Whitespace

Whitespace in ISL is defined as spaces or tab characters inserted into a program to either separate commands from their arguments or to improve program readability. For example, the programmer might find it easier to write:

```
numrow = ( ( ( num_guest - 1 ) * 2 ) + header )
```

instead of:

```
numrow=(((num_guest-1)*2)+header)
```

Whitespace can be placed anywhere in a script between two distinct language elements. Language elements are: commands, functions, system variables, user-defined variables, input/output specifiers, comments (//), relational and boolean operators, and commas. A language element is an indivisible piece of information which, if broken apart with whitespace, will generate an ISL "token" error. For a complete listing of error messages, please see *Appendix B*.

For example, the number -125.99 cannot be written as - 125. 9 9. The command **WaitForClear** cannot be written as **Wait For Clear**.

The table below and on the next page shows the incorrect and correct ways of using whitespace:

ISL Language	Incorrect Use of Whitespace	Correct Use of Whitespace
Within a variable name	num _ columns = 1	num_columns = 1
Within a command or function name	load ky bd macro 1:12	loadkybdmacro 1 : 12
Within a numeric value	- 19 8. 45	-198.45

ISL Language	Incorrect Use of Whitespace	Correct Use of Whitespace
Within a command format specifier	A 12 N 5 $ 12	A12 N5 $12
Within a comment delimiter	/ / comment...	// comment...
Within double-character relational operators	a < > b count < = 5 Joe > = Richard	a<>b count <= 5 Joe >= Richard
Between the @ and a System Variable name	@ TNDTTL @ YEAR	@TNDTTL @YEAR

The following table shows examples of where whitespace increases program readability:

No Whitespace	With Whitespace
var count[10]:N5	var count[10] : N5
display 1,2,"Room is ",room	display 1, 2, "Room is ", room
txmsg "CHG",guest,@tndttl+1.00	txmsg "CHG", guest, @tndttl + 1.00

Writing and Editing Scripts

Scripts should be composed in an ASCII text file and saved with the appropriate file naming convention discussed “Programming Simphony for SIM” on page 2-10.

Note: *There is no need to compile the script, or process it in any other way before the SIM can read the script.*

Avoiding Errors

To avoid errors when writing and editing scripts, follow these basic guidelines:

- Verify that the script was named using the correct conventions (see page 3-9). Also make sure it is in the proper directory.
- Check the Simphony database programming.
- Review the structure of the script.
    - Are all global variables declared at the beginning of the script?
    - Within each event, have all local variables been declared before issuing the first command?
    - Are the subroutines that are called in events also within the same script? Does the called subroutine have the same name as the actual subroutine?
- Look at the programming style. See “Script Writing Style” on page 3-9.
- Check that events correspond to the correct SIM Inquiry or SIM Tender keys initiating the event or tender event, respectively. For further details about creating these keys, refer to “Programming Simphony for SIM” on page 2-10.
- Check the script for syntax errors. For instance, make sure that if a command has a corresponding command ending the task, include it. For example, the **Event** command must be used with the **EndEvent** command, which should always be the last line of an event procedure.

Testing Scripts

Before using the SIM application in a live environment, Oracle Hospitality recommends testing it for errors first.

Detecting Errors in Logic

When the script is run, any errors in syntax are detected by the SIM, and an ISL error message is displayed. However, some errors in logic may not be caught by running the script.

Scripts make no assumptions, so execute the instructions exactly as specified. It is possible to run a script and detect no syntax-errors, but still have problems with the logic. Therefore, Oracle Hospitality recommends that each task be stepped through in the script.

Stepping Through the Script

Follow these steps to test the script:

- Print the script for reference.
- Mark through each step of the script as it is tested and correct it. This procedure helps track the steps that tested.
- Confirm that the script has the correct filename, to link it to the appropriate SIM interface, and that it is in the correct directory.
- Execute each event in the script. Remember, an event can be initiated in three ways: by pressing a SIM Inquiry key, by pressing a SIM Tender key, or by responding to a message sent by an interfaced third-party system.
    - Make sure that the SIM Inquiry or SIM Tender key pressed executes the correct event. The key should be linked to a corresponding event in the script.
    - Test whether communications are active between the two systems and that each system responds appropriately to messages sent to it by the other.
- Test each step of the event, such as the flow and logic of **If…EndIf** and **For** loop statements.
    - Check any assumptions made in the script. For instance, if a condition

must exist in order for the next step to occur, check the condition. The programmer might need to display an error message or perform another step until this condition is met.

- Check the logic of each task. For example, if the script collects data from the user before querying a third-party database, make sure that the script prompts the user for data entry before starting the query.
- Correct any detected syntax errors. The ISL error message will provide the number of the line in the script where the error is found.
- Verify inputs and outputs. For example, if the script calls for a coupon to be generated when an operator closes a guest check, test for this case.

Documenting Scripts

Oracle Hospitality highly recommends that a readme.doc file be created to store as a companion to the script. This document should include vital information pertaining to what is required to run the script.

README.DOC File Contents

The README.DOC file should contain the following:

- The SIM version for which the script was developed
- The type of interface (TCP-based, etc.) for which the script is intended
- The required database programming for tender/media and inquiry keys (their numbers must match their Event declarations)
- Any required touchscreen configurations
- Any required macro key programming
- Any other considerations
- A vendor contact name and telephone/fax number to use in case additional support is needed

The file below is an example of a readme.doc file that should accompany a script.

README.DOC File Name and Location

The documentation should be located in an ASCII-formatted file with the name *x...x*.doc, where *x...x* is the same as the script. For example, the readme.doc file for pms1.isl should be named pms1.doc. The readme.doc file should be placed in the *\Micros\Simphony\etc* directory.

Chapter 4

Using Variables

In This Chapter

For those users who are new to script writing, or need to familiarize themselves with script writing conventions, this chapter discusses some basic script writing concepts to apply when creating scripts.

Variables and ISL

As in other programming languages, the ISL supports the use of variables for holding information, such as integers or character strings, that may change from one ISL event to another.

Typically, variable results are stored in expressions like the following example:

```
variable_name = expression
```

An *expression* is a combination of variables and operators. Expressions may also include constants and functions.

For example, in the expression below, the size of a window is computed from the number of elements in a list:

```
window_length = ListSize + 2
```

The user variable *window_length* has a unique name defined by the user. This variable is compared, using the operator “equal to”, or =, to the expression on the right. The expression on the right contains a user variable called *ListSize*, which is added to the integer 2. The result of the expression on the right will be the value assigned to the user variable *window_length*: the *ListSize* plus 2.

This chapter describes how to use the language elements that allow variable information to be held and evaluate expressions of variables.

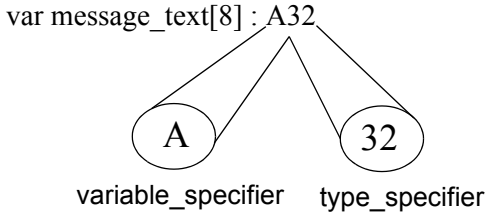
This chapter contains the following:

- A listing of the different kinds of data types variables can be in ISL
- A discussion of the mathematical operators used to evaluate expressions containing variables
- A discussion of the type of user variables supported by the ISL

Data Types

The ISL supports several kinds of data types, including **numeric**, **decimal**, **alphanumeric**, and **key** data types, to specify different kinds of variables and constants.

When a variable is declared using the **var**[1] command, its data type and size must also be declared. The type and size are referred to as the **variable_specifier** and **type_specifier**, respectively.



The table below lists the different kinds of data types and provides the abbreviations that must be used when declaring them.

Data Type	Abbreviation	Description	Example
Numeric	N*x*	The maximum size can be 32768 (N32768). However, only the first nine digits are significant when any arithmetic operation is performed.	If the variable were defined as `test:N12 = 123456789012`, then `test+1` would not evaluate correctly due to truncation.

1. For complete description of the **var** command, refer to page 7-184.

Data Type	Abbreviation	Description	Example
Decimal	$*x*	These variables are used for decimal amounts. Operator entries will assume a decimal place according to the currency's default setting, as specified in the *Currency* file; i.e., entering 1234 in the US will result in an amount of 12.34. They may comprise *x* digits, e.g., $4 in the US will support -99.99 to 99.99. The maximum size can be 32768 (N32768). However, only the first sixteen digits are significant when any arithmetic operation is performed.	If the variable were defined as `test:$18 = 12345678901 2345678`, then `test+1` would not evaluate correctly due to truncation.
Alpha-numeric	A*x*	These variables may include any non-control character, including punctuation marks. They may comprise *x* characters.	var name : a20
Key	key	This system variable is used for key press variables.	var keypressed : key

Example

The example below declare a 32-character alphanumeric variable and a four-digit room number variable, respectively.

```
event : 1

var message_text[8] : A32
var room_num : N4
.
.
.
```

Relational and Logical Operators

The mathematical operators described in this section are supported by the ISL. Before using these operators in a script, review each description carefully, as well as the "Operator Rules" on page 4-6.

For an explanation of the operand types (N*x*, $*x*, A*x*, and Key), please see "Data Types" on page 4-3.

Unary Operators

There are two unary operators:

Operator	Description	Example
-	Negation operator (a minus sign). This is used to negate an expression.	-3 -count -((count + 5) * -index)
NOT	Will negate the result of the expression. The NOT operator can be applied to expressions in the same way as the unary minus operator.	-3 -count -((count + 5) * -index)

Given that ISL expressions are true if they evaluate to a non-zero value and false if they are zero, the NOT operator will change non-zero values to 0 and 0 values to non-zero. The expression NOT 3 is valid and will evaluate to a 0.

The NOT operator is generally used in **If**, **ElseIf**, and **While** statements to control program flow.

The following is an example of a loop that looks for the end of a file:

```
while NOT feof( fn )
      .
      .
      .
endwhile
```

Binary Operators

The following table lists the available binary and logical operators in order of precedence (highest to lowest). AND and OR, the logical operators, have a lower precedence than all the binary operators.

Operation	Operator	Allowable Operand Types: Nx, $x, Ax, and Key
multiplication	*	**N***x*, **$***x*
division	/	**N***x*, **$***x*
modulus	%	**N***x*, **$***x*
plus	+	**N***x*, **$***x*
minus	-	**N***x*, **$***x*
bit-wise and	**&**	**N***x*
bit-wise or	**|**	**N***x*
equality	**=**	**N***x*, **$***x* **A***x*, **Key**
greater than or equal	**>=**	**N***x*, **$***x* **A***x*, **Key**
greater than	**>**	**N***x*, **$***x* **A***x*, **Key**
less than or equal	**<=**	**N***x*, **$***x* **A***x*, **Key**
inequality	**<>**	**N***x*, **$***x* **A***x*, **Key**
less than	**<**	**N***x*, **$***x*, **A***x*, **Key**
logical and	**AND**	**N***x*
logical or	**OR**	**N***x*

Operator Rules

Relational Operators

All relational operators produce a non-zero value if they are true, and 0 if they are false. For example:

```
result = 1 < 2            //true, result will be non-zero
result = 100 < 4          //false, result will zero
```

Logical (Boolean) Operators

The logical (Boolean) AND and OR operators treat their operand as either true (non-zero) or false (zero) values.

For example:

```
result = 5 AND 6
     //true, since 5 and 6 are both non-zero
result = 5 AND 0
     //false, since 0 is false
result = 0 OR 0
     //false, since neither one is true
result = 0 OR 5
     //true, at least one value is non-zero
```

Precedence

- ISL expression operands are evaluated from left to right until the end of the expression is reached. For example, in the following expression, `1 + 5 + 2`, the 1 is added to the 5, equaling 6, then 2 is added to 6, resulting in 8.

- When expressions mix operators (i.e., + and *), then ISL will use the precedence table to determine which subexpression within the expression will evaluate first. In the following example, the `*` operator has higher precedence then the + operator, therefore, the last two operands will be combined first, even though they are not the first in the expression: `1 + 5 * 2`

  The result is 11 (1 + 10), rather than 12 (6 * 2). The precedence rules are used for all operators. Since the < operator has greater precedence than the OR operator, then in the following expression: `a < 1 OR b > 3`

  `a < 1 and b > 3` are evaluated first, and both results are combined with the OR operator.

Overriding and Clarifying Precedence

The parentheses can be used to override the default precedence rules. Parentheses are used for two reasons:

- To override the default expression evaluation
- To clarify the expression

Subexpressions enclosed in parentheses ***always*** override the operator evaluation. For example, in the following expression:

```
(1 + 5) * 2
```

the `1 + 5` is evaluated first since it is within parentheses, even though `the multiplier *` has higher precedence.

```
The following expression:

a < 1 OR b < 3
```

can be rewritten as:

```
(a < 1) OR (b < 3)
```

It is good practice to always place parentheses around subexpressions, to reduce programming errors and to make scripts more easily understood and maintained.

For example, the following expression:

```
offset + width * 2 <= w_width / stlen + 1
```

is equivalent to:

```
(offset + (width * 2) )  <=( ( w_width / stlen)  + 1)
```

The second expression is clearer in its intent.

User Variables

User variables are defined by the interface designer and may be used to get operator input, such as customer name and address information, a room number, etc. They can also reference an entry in a message received over a SIM Interface.

Declaring User Variables

Declare variables just as is in C, but in this case, use the **var** command to do this. These variables are given a value by an operator, or by the interface, in a response message. Example:

```
event inq : 5

var rowcnt : n3
.
.
.
```

Guidelines

- The variable name must begin with a letter A-Z, a-z, or the underline character (_).
- The first character cannot be a number. It may subsequently include any character in the range A-Z, a-z, 0-9, or the underscore _ character.
- Initially, numeric variables should always be set equal to 0.
- String variables should initially equal a null string " ".

Remember that when declaring SIM numeric or decimal variables, large variables used in mathematical operations may be truncated. All operations involving numeric variables use only the first nine digits, and decimal variables use only the first sixteen digits.

The scriptwriter can still declare and assign large variables. For example, it is still valid to create an N10 variable that will hold a telephone number, or an N16 variable that will hold an access code. However, any non-relational expressions may cause truncation and yield the wrong answer.

Local and Global Variables

Variables can be declared either globally or locally.

Global Variables

If a variable is declared outside of an Event procedure, it is considered global. The variable is called global because it can be referenced by any event or subroutine in the script. As a result, the ISL must maintain the contents of the variable the duration of a script. The variable is then reset at the start of a new transaction (i.e., when tendering a check). Also, since the variable will be used by other events and subroutines in the script, a global variable needs to be declared only once at the top of the script before any events or subroutines.

Local Variables

Conversely, a variable is considered local when it is declared inside an event procedure. Local variables are only maintained while the event procedure is being executed; executing the **EndEvent** command, or any other command that stops the script, purges the local variable from the event procedure. Thus, local variables must be declared within event procedures.

Local Variables Used by Subroutines

Local variables declared by a parent event procedure are visible within a child subroutine. This functionality is possible because a local variable is accessible to the event in which it is included, and to any child subroutine called by the parent event. Moreover, the contents of a local variable, declared in the parent event, can be changed by the operation of a subroutine called by the same parent event. Consequently, the new contents of the local variable are retained when the subroutine is complete.

Array Variables

Arrays of variables can also be declared by including an array size with the declaration. The syntax for an array variable is:

Var *variable_name[array_size]:variable_specifier*

The example below declares an array of strings named message_text, containing eight elements, each 32 characters in length.

```
var message_text[8]:A32
```

Note: *The use of brackets in the example above does not denote an optional entry, but is actually part of the syntax.*

For more information on the **var** command, please see page 7-184.

Variable Size Variables

It is possible to declare a variable with a size that is defined by an expression, rather than a hard-coded number. If the SIM encounters an open parenthesis immediately following the type of the variable, it will assume that an expression follows, defining the variable's size. For example, the following commands have the same effect:

```
var window_width : N30
var window_width : N(15 + 15)
```

This feature is useful for declaring variables whose size is not known until run-time. Example:

```
var longest_str : A(max_str_len)
```

Using List Arrays and Records

The application data message contains the information that the SIM sends to and receives from an interface. The message consists of a set of ASCII fields, separated by the ASCII field separator (1CH).

In many cases, the number of variables to be sent by the SIM or the interface is not known until run-time. For example, a script may query the PMS for a list of guests whose last name starts with "SMITH." The PMS may respond with two names, or with ten names, depending on who is in the hotel at the time.

In order to send and receive variable amounts of data, ISL uses two methods: list arrays and records.

List Arrays

The SIM provides more than one method for sending and receiving variable amounts of data within one message. The simplest method is to send a list. A list consists of a *list_size* and the *array_variable* that contains the list. The *list_size* is any user-defined integer variable. The *array_variable* is any user-defined array, as shown on page 4-11.

Specifying a List Array

A list is specified for an **RxMsg** or **TxMsg** command by using empty array brackets ([]) after the array name.

Example: **Rxmsg** *list_size, list[]*

When specifying a list array, follow these guidelines:

- The variables used for the list size and the list are user-defined.
- The list size variable should always precede the list array variable.
- Array system variables (e.g., @DTL_OBJNUM) cannot be used as list arrays.
- The values in the list[ ] should be formatted into as many lines as are specified in *list_size*. In the example below, *list_size* is 5, thus five values from the list[ ] will be formatted if these ISL statements are executed.

```
var list_size : N5
var list[ 10 ] : A20
txmsg list_size, list[]
```

Using the same example, if the [fs] symbol stands for the field separator, then the following lines will create these messages:

```
list[ 1 ] = "L1"
list[ 2 ] = "L2"
list[ 3 ] = "L3"
list[ 4 ] = "L4"
list[ 5 ] = "L5"

txmsg 3, list[]            //3[fs]L1[fs]L2[fs]L3
txmsg 5, list[]            //5[fs]L1[fs]L2[fs]L3[fs]L4[fs]L5
txmsg 0, list[]            //0
```

- The PMS should read the first value in the message and receive that many elements from the rest of the message. The **RxMsg** command

reads the data from the message in a similar manner. Example:

```
var listsize : N5
var list[ 10 ] : A20


//If message from PMS is: 2[fs]L1[fs]L2, then
//listsize = 2
//list[1] = "L1"
//list[2] = "L2"

rxmsg listsize, list[]

//If message from PMS is: 4[fs]L1[fs]L2[fs]L3[fs]L4, then
//listsize = 4
//list[1] = "L1"
//list[2] = "L2"
//list[3] = "L3"
//list[4] = "L4"

rxmsg listsize, list[]
```

In another example, the script collects a guest name from the operator, transmits the guest name to the PMS, receives a list of names from the PMS, and displays the list in a window:

```
event inq : 1
     var name : A20
     input name, "Enter guest name"
     txmsg "GST_INQ", guest
     waitforrxmsg
endevent

event rxmsg : GST_RSP
     var guest_count : N5        //declare list size
     var guest_name[10] : A20    //declare list
     rxmsg guest_count, guest_name[]
                                 //receive up to 10 names
                                 //open up window


     window guest_count, 25, "Guest List"
     listdisplay 1, 1, guest_name //display names
     waitforclear
endevent
```

- Lists can be intermingled with other non-list variables, as well as other lists. In the following example, one single variable and two lists, each with its size variable, are received from the PMS:

```
rxmsg status, guest_count, guest_name[], action_count,\
action_list[]
```

- For each list, only one array may be assigned. It is possible, for

example, for the PMS to send not only the guest name, but also the room number. One way to handle this would be to receive two lists within one message:

```
event rxmsg : GST_RSP
      var guest_count : N5
      var guest_name[10] : A20h
      var guest_room[10] : N5
      rxmsg guest_count, guest_name[], guest_count,\
      guest_room[]
            .
            .
endevent
```

Implicit List_Sizes

There are occasions where the script writer may desire to use lists, but does not want to actually specify the *list_size*, since the *list_size* is not specified in the data. For example, assume that each line in a file contains 20 fields. In order for the ISL to read each line, a separate variable must be specified in the **Fread** command for each field.

```
var n[20]
fread fn, n[1], n[2], n[4], n[5], n[6], n[7], n[8],\
   n[9], n[10], n[11], n[12], n[13], n[14], n[15], n[16],\
   n[17], n[18], n[19], n[20]
```

The ISL provides a method for specifying *list_sizes* implicitly for those cases when the data does not contain a *list_size*. An implicit *list_size* is identified by a pound symbol (#) placed before the *list_size*. Alternatively, the example above could be written as:

```
var n[20]
fread fn, #20, n[]
```

Only integer expressions may be placed in the implicit *list_size* field. Integer variables may also be used.

```
var n[20]
var size : N5 = 20
fread fn, #size, n[]
```

Implicit *list_sizes* may be used anywhere standard *list_sizes* may be used.

Records

ISL also provides a more powerful, yet more complicated, syntax for specifying variable amounts of data called **records**. In this format, the variables following the list size are considered to be in groups of records.

The syntax for receiving this type of information is:

```
listsize, list1[] : list2[] : list3[] ...
```

The colon separates fields within a record and must be used when specifying records.

For example, if the PMS received an inquire on SMITH, then it may want to group the data as follows:

```
3[fs]Smith[fs]1423[fs]Smithers[fs]1827[fs]Smithson[fs]1887
```

Note that each record consists of a name followed by a room number.

To receive the message above, use the following statement:

```
rxmsg count, name[] : room[]
```

In this case, the variables would be set as follows:

```
count = 3
name[1] = "Smith"
name[2] = "Smithers"
name[3] = "Smithson"
room[1] = 1423
room[2] = 1827
room[3] = 1887
```

The same format would be used for transmitting data.

A list specification is a special case of a record format, where each record consists of one element.

Promotion

ISL allows the programmer to freely combine and assign variables of the different types. For example, it is possible to add a string and an integer, and assign it to a decimal value.

```
var amt : $10 = 12 + "25" + 100.45
```

Whenever two variables and/or constants are operated upon with an operator, and they are not the same types, one will be “promoted” (have its type changed) before the operation takes place.

Strings promote to integers and integers promote to decimal values. A final promotion occurs when the expression is assigned to a variable. Therefore, the expression is promoted (or demoted) to the variable type.

For example, if the following variables are declared:

```
var string : A20
var integer : N10
var decimal : $10
```

Then the following statements are assigned within an Event procedure, the statements would be equivalent to:

Assignment	Equivalent To
string = "12" + 35	string = "47"
string = "14.15" + 2	string = "16"
string = "14.15" + 2.00	string = "16.15"
integer = "14" + 12.5	integer = 26
integer = "14.5" + 12.5	`integer = 27` `expression was real, and then demoted to integer`
decimal = 12.23 + 1	decimal = 13.23
decimal = "12.23" + 1	`decimal = 13.00` `"12.23" + 1 yields an integer 13, which is then assigned to decimal`

Strings are converted to integers by using the first digits in the string field. “12.35” converts to an integer 12, since “.” does not belong in an integer. “12NUM” also converts to 12. Therefore, it is legal to write:

```
integer = "ABC123"          // integer = 0
```

Only relational operators are allowed between strings; see “Relational and Logical Operators” on page 4-5.

Correct:

```
integer = "12" > "35"
```

Incorrect:

```
integer = "12" + "35"
```

Chapter 5

ISL Printing

In This Chapter

This chapter contains an introduction to the ISL command and system variables that facilitate output to print devices.

Getting Started with ISL Printing

Printing in ISL is accomplished using the **StartPrint**, **PrintLine**, and **EndPrint** directives (or their variants).

- Backup printers may be specified;
- Printouts can end with Form Feeds or not; and
- Printing text in double-wide characters and red ink is also supported.

This chapter focuses on how to start and direct print jobs to printers. And in performing these tasks, the other options available are covered as well, including detecting the status of print jobs, redirecting print jobs to ISL-defined backup printers, and defining a reference line for print error messages.

ISL Print Commands and System Variables

All of the commands and system variables associated with ISL printing are discussed. For complete descriptions, including syntax and examples of all commands discussed in this section, refer to “ISL Commands” on page 7-1. For more information about system variables and their use, refer to “ISL System Variables” on page 6-1.

Starting an ISL Print Job

Print jobs include guest checks, customer receipts, validation chits, local backup printing, remote order printing, as well as journal printing.

These types of print jobs are initiated by variations of the **StartPrint** command described in this section. These variations are designed to accommodate the ability to detect whether a print job completed successfully by using the @PRINTSTATUS system variable.

ISL StartPrint Commands

Printing in ISL is started using the command in the table below. The **StartPrint** command is used to print to a standard remote printer. These printers will print a maximum of 32 characters per line and understand special formatting.

Command	Sets @PRINTSTATUS	Form Feed	Backup Printer	Reference Line
StartPrint…EndPrint[FF/NOFF]	♦[1]	♦	♦	♦

1. Always set to *Y*.

Extended Printing and Printing Binary Data

Extended and Binary printing is possible using a certain set of ISL command and system variables. See the following:

- Print_Header Event
- Print_Trailer Event
- @HEADER System Variable
- @TRAILER System Variable
- PrintLine Command

Form Feeds

The EndPrint command can issue three types of form feeds:

- EndPrint
- EndPrintNOFF
- EndPrintFF

The EndPrint command is used when the default behavior for formfeeding at the end of a print session should be used. For journal printers, there is no formfeed. For all other printers a form feed is used.

The EndPrintNOFF command is used to prevent a formfeed being sent at the end of a print job.

The EndPrintFF command is used to always force a formfeed at the end of a print job.

Backup Printing and Reference Lines

The commands used for print jobs also allow for the specification of a backup printer in case the print job fails. This backup printer overrides any backup printer already programmed in the database for the specified printer. In addition, the text can be specified, called a reference line, to appear in printer error messages returned when print jobs fail.

Specifying an ISL Printer

When specifying a **StartPrint** command, there are two options for defining a printer: the object number or a system variable.

Using DTENs

All printing requires that a printer be specified by the **StartPrint** command. A printer can be identified either by its device table entry number (DTEN), which is the device's object number, or by a system variable. For example, if the print job goes to device 8 in the *Printers* module, using the object number, the **StartPrint** command would be written as follows:

```
StartPrint 8
```

Although using this method is valid, it has a primary disadvantage. For instance, the destination printer for ISL print jobs may differ for each workstation. So if the object number had to be hardcoded, as it is in the example above, then a different script would be required for each workstation. Thus, each script would need to specify the printer to which each workstation must print, limiting the flexibility of the script.

Considerations

- Although it is legal to specify a printer object number of 0, all print jobs printing to 0 will not print—anywhere.
- An invalid object number value (-1) will generate an ISL error.

Using System Variables

A more efficient method of specifying printers is through the use of system variables. These system variables return the value of the object number of the printer. For example, the @CHK system variable will return the object number of the printer defined as the Check Printer for that workstation in the database.

Since each workstation will have a different entry for its Check Printer, the ISL command “startprint @chk” will specify a different printer for each UWS. For example, on Workstation #1, @CHK is 8 but, on Workstation #2, @CHK is 12. Using a system variable instead of the object number to specify a printer, means that each workstation can use the same script, yet still print to different printers.

ISL Printer System Variables

The printer system variables available in ISL include the following:

System Variable	Description
@CHK	Guest Check Printer
@RCPT	Customer Receipt Printer
@ORDR[1…15]	Remote Order or Local Backup Printer
@VALD	Validation Printer

A table describing all options available with each valid ISL printer is provided with the detail description of the **StartPrint** command. See “StartPrint...EndPrint[FF/NOFF]” on page 7-163.

Using Print Directives

The ISL Print Directives consist of one-byte values sent to the printer, defined by the **StartPrint** command, to change the print type of the *expression* that follows it. These directives are similar in function to standard parallel printer escape sequences: Each print directive is a non-printable character and is included in the print data sent to the printer.

The Printline Command

Print Directives are actually system variables, and are arguments of the **Printline** command. This command allows the ISL to print information provided as a text string or variable.

The **Printline** command prints a line on the selected printer defined by the **StartPrint** command, which must be issued before **Printline**. Depending on the print directives specified in the Printline statement, the expression will print in double-wide characters or red ink.

Print Type System Variables

Several system variables that evaluate to these print directives are provided by the ISL to facilitate printing expressions in double-wide characters and in red ink. These directives are described in the table that follows.

Print Directive	Description
@DWON	Prints the following text or variable fields double-wide. Single- and double-wide characters may be mixed on the same line.
@DWOFF	Prints the following text or variable fields double-wide. Single- and double-wide characters may be mixed on the same line.
@REDON	Prints an entire line in red. It is not possible to mix red and black characters on the same line. All new lines default to black.
@REDOFF	Returns printing to default ink (blue, black, etc.) All new lines default to black.

Considerations

- Print directives (@DWON and @DWOFF) may be inserted between *expressions*, but only affect the expression to the right.
- The print directives are reinitialized at the end of each printed line.
- If no print directives are specified, then printing will be in black and single-wide.

Example

The following example illustrates how the same expression can be printed in four different ways by using a combination of these print directives:

```
Printline "Print line"                    //prints in black
Printline @redon, "Print line"            //prints in red
Printline @dwon, "Print line"             //prints in black,
                                          // double-wide
Printline @dwon, @redon, "Print line"     //prints double-wide
                                          // in red ink
```

Print Directives and Subroutines

Since the print directives are normal ISL strings, they can be passed as arguments to subroutines. The following example prints an array of data and displays a header using a print directive passed in as a parameter to the subroutine:

```
sub print_list( var printer : N9, var listsize : N5,\
     ref list[], ref header_string, var directive : A1 )

   if printer = 0
      errormessage "Printer dten is 0. Cannot print."
      exitcancel
   endif

   startprint printer
      printline "----------------------------"
      printline directive, header_string
      printline "----------------------------"
      listprint listsize, list
   endprint

endsub
```

The subroutine could be invoked in the following fashion:

```
event inq : 1
.
.
.
print_list( @chk, sz, data[], "NORMAL HEADER", @redoff )
.
.
.
print_list( @rcpt, sz, data[], "RED HEADER", @redon )
endevent
```

Backup Printing

Whenever the **StartPrint** command is issued, a print job will occur. If the print job is unable to complete successfully, it will go to the backup printer defined in the Simphony database for each printer type (i.e., @chk, @rcpt, ...).

However, there are instances when a backup printer different from the one defined in Simphony should be specified. To accommodate these instances, the **StartPrint** command accepts an optional second argument. This optional argument specifies the object number of the backup printer, overriding the backup printer programmed in the Simphony database.

For example, if the Check Printer is normally backed up by the Customer Receipt Printer but the script requires that a print job to the Check Printer back up to the Order Printer, the following command should be issued:

```
STARTPRINT @chk, @ordr[1] // back up to @ordr[1] instead of @rcpt
```

Considerations

The SIM will only route the print job to the backup printer defined in the command syntax if the primary printer specified is a system variable. Otherwise, the ISL does not know to which printer type the job should be re-routed. For example, assume that the @CHK system variable equals 2.

```
STARTPRINT @chk       // ISL can determine backup printer
STARTPRINT 2          // ISL cannot determine backup printer
```

In the first line, the SIM will correctly determine the backup for the Check Printer, since “@chk” is explicitly specified. In the second line, the number 2 is used instead, and ISL cannot correctly determine the backup printer, and so, no backup is used.

Reference Strings

Whenever the SIM is performing a print job and an error occurs during printing (paper out, door open), an error message will appear on the display of the workstation explaining the error. Included in this error message is a line of text identifying the print job.

Normally, ISL will leave this line blank. However, this reference line can be specified in both the **StartPrint** commands.

```
STARTPRINT @chk                  // use default backup
STARTPRINT @chk, @rcpt           // no reference line

STARTPRINT @chk, @rcpt, "Printing Customer Coupon"//ref line
```

Chapter 6

ISL System Variables

In This Chapter

This chapter summarizes all ISL system variables in an A-Z reference format.

System Variables

System variables return status information from Simphony, Windows® Status flags, or the PMS System, etc., as well as provide access to transaction totals and other transaction parameters.

The following can be accomplished using the system variables that the ISL supports:

- Access the system transaction variables and totals information and
- Set certain operational parameters.

This chapter contains a detail description of each system variable.

System Variable Summary

For quick reference, a summary of system variables in alphabetical order and in order by category of function begins on page 6-7.

Specifying System Variables

Review this section to determine the guidelines to follow when specifying system variables.

Specifying System Variables

This section contains quidelines that should be followed when specifying system variables in ISL statements.

Guidelines for Specifying System Variables

Follow the guidelines below when specifying system variables:

- Since the names for system variables are reserved, do not declare other user variables using the same name.

  The example below is incorrect because the local variable *ccnumber* has the same name as the system variable @CCNUMBER, which returns a credit card account number.

```
event : 1
  var ccnumber : A16
.
.
.
```

  This problem can be corrected by replacing *ccnumber* with *account_num*, which is a user variable that represents the credit card account number, but is not a system variable.

```
event : 1
  var account_num : A16
.
.
.
```

- Always precede each system variable with an At @ character. Example: `@SI[1]`

- Never put spaces between the At @ and the system variable name.

  Correct: `@SI[ 1 ]`

  Incorrect: `@ SI[ 1 ]`

- Before using a system variable, review the detail description carefully for any special considerations, such as:

    - The majority of system variables must be used in conjunction with other commands, functions, or other system variables. For instance, the @DWON system variable can only be used with the **Printline** command.

    - Some system variables are only valid within a certain event. For example, @CCNUMBER and @CCDATE will only return valid values if issued from within a tender/media event referencing a credit card tender type.

    - A strategically placed system variable may or may not be required within the script. For example, the @WARNINGS_ARE_FATAL system variable must be placed at the top of the script. But the @LINES_EXECUTED system variable can be placed anywhere in a script.

- Just as there are user variables that can be specified as arrays, there are array system variables. Array system variables require a reference to an array index. "Using an Index to Specify System Variables" below describes how to issue these types of system variables.

Using an Index to Specify System Variables

Specifying array system variables is the same as specifying user-declared array variables. Array references in ISL take the form:

```
<array name> [ <expression> ]
```

where <*array name*> can be either a user or system array variable, and <*expression*> (i.e., the index) can be a user variable, another system variable, a constant, a string, a function, or an equation.

As long as the array index evaluates to an expression within the array limits for the system variable, the index can be specified as a user variable, another system variable, a constant, a string, a function, or an equation. In the following examples, three different references evaluate to 3:

```
@si[ 3 ]                    // constant
@si[ 6 - 3 ]                // equation
@si[ ( index * 2 ) - 1 ]    // equation using user variable,
                            //     where index = 2
```

Array Subscripts

The difference between system array variables and user-declared array variables, is that system array variables already have been declared and filled with the corresponding information. Consequently, there is no need to declare the subscripts of the array.

For example, the user-declared array variable must be declared as follows:

```
var myarray[5] : A20
   myarray[1] = "mytest"
   myarray[2] = "mytest"
   myarray[3] = "mytest"
   myarray[4] = "mystes"
   myarray[5] = "mytest"
```

But a system array variable can be specified as:

```
@si[3]
```

For a system array variable, the variable data type and size, and subscript are assumed by the ISL to reference Sale Itemizer #3. If executed by a script, this system array variable would return the totals posted to this sales itemizer on the current guest check.

Array Index Limits

The array index limits are included for system variables with the detail description of each system array variable. These limits vary depending on the system variable.

If the array index exceeds the limit when referencing the system variable, an error will occur. For example, below are invalid limits for the @SI system array variable:

```
@si[10]                    //incorrect
@si[-10]                   //incorrect
```

The array index for `@SI` must evaluate to an index between 1 and 16.

All array indices start at 1, and not 0. For example, `@si[0]` will generate an error.

Embedded Index vs Array-Index

System variables that require an index (e.g., DTL_*, @SI, etc.) can be referenced in two ways: by an embedded index or array-index. Both of these methods enable older versions of SIM scripts to maintain compatibility with the ISL. In early versions of the ISL, there was support only for the embedded index method.

If using the embedded index to maintain older SIM scripts (Version 1.01S or earlier), the desired index is placed immediately after the system variable.

Example:

```
@SI2, @TAX1, @TXBL1, @DTL_STATUS9
```

In scripts compatible with Version 1.01T or higher of the ISL, the array-index method is the preferred way to specify system array variables. The array-index requires a to reference the index as an array.

Example:

```
@SI[ 2 ], @TAX[ 1 ], @TXBL[ 1 ], @DTL_STATUS[ 9 ]
```

Note: *The embedded-index method remains in ISL to retain compatibility with older scripts and should not be used with new scripts.*

System Variable Summary

For quick reference, this section contains an alphabetical summary of all ISL system variables.

Note: *ISL variables are listed by category in Appendix C.*

Variable Name	Field/Parameter
@ACTIVE_LANGID	ID Number of Currently Selected Language
@ADDXFER_CHK_FROM	Check Number of the Check Being Transferred From
@ADDXFER_CHK_TO	Check Number of the Check Being Transferred To
@ADDXFER_GRP_FROM	Table Group Number of the Check Being Transferred From
@ADDXFER_GRP_TO	Table Group Number of the Check Being Transferred To
@ADDXFER_RVC_FROM	Rev. Center Number of the Check Being Transferred From
@ADDXFER_RVC_TO	Rev. Center Number of the Check Being Transferred To
@ADDXFER_TBL_FROM	Table Number of the Check Being Transferred From
@ADDXFER_TBL_TO	Table Number of the Check Being Transferred To
@ALPHASCREEN	Alpha Touchscreen
@AUTOSVC	Auto Service Charge
@BEVERAGE_REQD	Prompt User For Beverages
@CCDATE	Credit Card Expiration Date
@CCNUMBER	Credit Card Account Number
@CENTER	Center Column in ISL-defined Window
@CHANGE	Change Due
@CHECKDATA	Facsimile of Check
@CHGTIP	Charged Tip
@CHK	Guest Check Printer

Variable Name	Field/Parameter
@CHK_OPEN_TIME	Date and Time Check Opened
@CHK_OPEN_TIME_T	Current Check Open Time
@CHK_PAYMNT_TTL	Current Check Payment Total
@CHK_TTL	Current Check Total
@CKCSHR	Guest Check Cashier Number
@CKCSHR_NAME	Guest Check Cashier's Name
@CKEMP	Check Employee
@CKEMP_CHKNAME	Check Employee's Check Name
@CKEMP_FNAME	Check Employee's First Name
@CKEMP_LNAME	Check Employee's Last Name
@CKID	Guest Check ID
@CKNUM	Check Number
@CLIENT_ONLINE	Determine if SAR Workstation is Online
@DAY	Current Day of Month
@DBVERSION	Current Database Version
@DETAILSORTED	Detail Sorting Status
@DSC	Discount Total
@DSC_OVERRIDE	When a manual discount is entered, a SIM 'Discount' script can decrease the amount of the discount by setting this variable to the desired discount amount
@DSCI	Discount Itemizer Value
@DTL_CAACCTINFO[ ]	Credit Authorization Account Information
@DTL_CABASETTL[ ]	Credit Authorization Base Total
@DTL_CAEXPDATE[ ]	Credit Authorization Expiration Date
@DTL_CATIPTTL[ ]	Credit Authorization Tip Total
@DTL_CATMEDOBJNUM[ ]	Credit Authorization Tender/Media Object Number
@DTL_DEFSEQ	Definition Sequence of Detail Item
@DTL_DSC_EMPL[ ]	Employee who is getting the employee meal discount for the specified detail entry
@DTL_DSCI[ ]	Menu Item Detail Class Discount Itemizer Value
@DTL_FAMGRP[ ]	Menu Item Family Group

Variable Name	Field/Parameter
@DTL_INDEX	Index of the detail which fired the SIM event
@DTL_IS_COND[i]	Determines if a Guest Check Menu Item is a Condiment
@DTL_IS_VOID[i]	When set (non-zero), the specified detail is a Voided Item
@DTL_MAJGRP[ ]	Menu Item Major Group
@DTL_MLVL[ ]	Main Menu Level of Detail Item
@DTL_NAME[ ]	Name of Detail Item
@DTL_OBJNUM[ ]	Object Number of Detail Item
@DTL_PLVL[ ]	Price Level of Detail Item
@DTL_PMSLINK[ ]	PMS Link of Detail Item
@DTL_PRICESEQ[ ]	Price Sequence Number of Detail Item
@DTL_QTY[ ]	Quantity of Detail Item
@DTL_SEAT[ ]	Seat Number of Detail Item
@DTL_SLSI[ ]	Menu Item Detail Class Sales Itemizer Value
@DTL_SLVL[ ]	Sub-menu Level of Detail Item
@DTL_STATUS[ ]	Status of Detail Item
@DTL_SVC_LINK[ ]	Stored Value Card Link
@DTL_SVC_TYPE[ ]	Stored Value Card Type
@DTL_SVCI[ ]	Menu Item Detail Class Service Charge Itemizer
@DTL_TAXTTL [ ]	Returns the Total Tax Amount for the Detail
@DTL_TAXTYPE[ ]	Active Tax Types
@DTL_TTL[ ]	Total of Detail Item
@DTL_TYPE[ ]	Type of Detail Item
@DTL_TYPEDEF[ ]	Returns the Detail Item Type Definition
@DWOFF	Double-wide Characters OFF
@DWON	Double-wide Characters ON
@EMPLDISCOUNT	In a discount event, this variable is the number of the employee discount

Variable Name	Field/Parameter
@EMPLDISCOUNTEMPL	In a discount event, this variable is the employee number of the discount receiving the employee discount
@EMPLOPT[ ]	SIM Employee Options #1-#8
@EPOCH	EPOCH Time
@EVENTID	String that represents the event ID
@EVENTTYPE	String that represents the event type
@FIELDSTATUS	Data Entry Field Status Flag
@FILE_BFRSIZE	User Definable Variable
@FILE_ERRNO	Standard Error Number Value
@FILE_ERRSTR	Standard Error String based on @FILE_ERRNO
@FILE_SEPARATOR	Field Separator for File I/O Operations
@FILTER_ACTIVE	Seat Filter Active
@FILTER_MASK	Current Seat Filter Mask
@GRPNUM	Table Group Number
@GST	Guest Count
@GSTRMNG	Guests Remaining after Proration
@GSTTHISTENDER	Guest Count Associated with Split Tender
@GUID	The GUID of the Current Check
@HEADER	Print Header from Print_Header Event
@HOUR	Current Hour of Day
@IGNOREPRMT	Bypass general operator prompts with the Enter key
@INEDITCLOSEDCHECK	Edit Closed Check Entry
@INPUTSTATUS	User Input Status Flag
@INREOPENCLOSEDCHECK	Reopen Closed Check Entry
@INSTANDALONEMODE	Determine if SAR Workstation is Offline
@ISUNICODE	Determines if Unicode Characters are Supported
@KEY_CANCEL	Cancel Key
@KEY_CLEAR	Clear Key
@KEY_DOWN_ARROW	Arrow Down Key

Variable Name	Field/Parameter
@KEY_END	End Key
@KEY_ENTER	Enter Key
@KEY_EXIT	Exit Key
@KEY_HOME	Home Key
@KEY_LEFT_ARROW	Arrow Left Key
@KEY_PAGE_DOWN	Page Down key
@KEY_PAGE_UP	Page Up key
@KEY_RIGHT_ARROW	Arrow Right key
@KEY_UP_ARROW	Arrow Up key
@LANG_ID[]	ID Numbers of Defined Languages
@LANG_NAME[]	Language Names for Defined Languages
@LASTCKNUM	Last Check Number Assigned to Guest Check
@LINE	Current Line Executed in Script
@LINE_EXECUTED	Lines Executed in Script
@MAGSTATUS	Magnetic Card Entry Status Flag
@MAXDTLR	Maximum Size of @TRDTLR
@MAXDTLT	Maximum Size of @TRDTLT
@MAX_LINES_TO_RUN	Maximum Lines of Script to Execute
@MINUTE	Current Minute
@MONTH	Current Month
@NUL	A binary 0 should be sent when printing binary data
@NUMOPNCHK	The count of Open Checks per Revenue Center
@NUMDSC	Active Discounts
@NUMDTLR	Number of Detail Entries this Service Round
@NUMDTLT	Number of Detail Entries for Entire Transaction
@NUMERICSCREEN	Numeric Touchscreen
@NUMLANGS	Number of Languages
@NUMSI	Active Sales Itemizers

Variable Name	Field/Parameter
@NUMSVC	Active Service Charges
@NUMTAX	Active Tax Rates
@OBJ	Object number of the detail item for the event
@OFFLINELINK	Used to link to an offline PMS system
@OPENCHECK_EMPOWNER	Object Number of the employee that owns the Open Check
@OPENCHECK_GUID	Open Check GUID (unique identifier)
@OPENCHECK_NUMBER	Open Check Number
@OPENCHECK_OPENTIME	Open Check Date and Time that the check was begun
@OPENCHECK_ORDERTYPE	Open Check Order Type ID number
@OPENCHECK_TOTAL	Open Check Total Amount
@OPENCHECK_WSOWNER	Object Number of the Workstation that currently owns the Open Check
@ORDERTYPE	Order Type
@ORDERTYPENAME	Active Order Type Name
@ORDR[ ]	Remote Order or Local Order Printer
@OS_PLATFORM	1 - Windows® CE 3 - Win 32
@PICKUPLOAN	Value of the pickup or loan amount
@PLATFORM	Hardware Platform
@PMSBUFFER	PMS Message
@PMSLINK	Revenue Center PMS Link
@PMSNUMBER	Object Number of PMS
@PREVPAY	Previous Payment
@PRINTSTATUS	Print Status Flag
@PROPERTY	The Property Number of the Workstation
@PRORATETND	Calculate prorated values
@QTY	Quantity of the detail item for the event
@RANDOM	Returns a random value between 0 and 2^{32}-1
@RCPT	Customer Receipt Printer

Variable Name	Field/Parameter
@REDOFF	Red Ink OFF
@REDON	Red Ink ON
@RETURNSTATUS	Transaction Item Return Indicator
@RVC	Revenue Center Number
@RVC_NAME	Current Revenue Center Name
@RVCSERIALNUM[ ]	Revenue Center Sequence Number
@RXMSG	Name of Return Message
@SEAT	Active Seat Number
@SECOND	Current Second
@SHOW_PMS_MESSAGES	PMS Status Flag
@SI[ ]	Sales Itemizers
@SIGCAPDATA	Signature Capture Data
@SIMDBLINK	Links to the SIMDB DLL to the database
@SRVPRD	Serving Period
@STRICT_ARGS	Strict Arguments
@SVC	Service Charges
@SVCI	Service Charge Itemizer Value
@SYSSERIALNUM[ ]	System Sequence Number
@SYSTEM_STATUS	Shell Return Status
@TAX[ ]	Tax Collected
@TAXRATE[ ]	Tax Rate
@TAXVAT[ ]	Returns the Value Added Tax Amount for Tax Rate “X”
@TBLID	Table ID
@TBLNUM	Table Sequence Number
@TMDNUM	Tender/Media Number
@TNDTTL	Tender Total
@TRACE	Output Line of Script to 8700d.log
@TRAILER	Print Trailer from Print_Trailer Event
@TRAININGMODE	Training Mode Status Flag
@TRCSHR	Transaction Cashier Number

Variable Name	Field/Parameter
@TRDTLR	Transaction Detail of Current Service Round
@TRDTLT	Transaction Detail of Entire Check
@TREMP	Transaction Employee
@TREMP_CHKNAME	Transaction Employee's Check Name
@TREMP_FNAME	Transaction Employee's First Name
@TREMP_LNAME	Transaction Employee's Last Name
@TTL	Amount of the detail item for the event
@TTLDUE	Total Due
@TXBL[ ]	Taxable Sales Itemizers
@TXEX_ACTIVE[ ]	Checks if the Tax is Exempt at the Specified Level
@USERENTRY	Data Entered Before SIM Inquiry Key Activated
@VALD	Validation Chit Printer
@VARUSED	Used Variable Space
@VERSION	SIM Version Number
@VOIDSTATUS	Transaction Item Void Indicator
@WARNINGS_ARE_FATAL	Strong Checking
@WCOLS	Number of Columns in ISL-defined window
@WEEKDAY	Day of Week
@WROWS	Number of Rows in ISL-defined window
@WSID	User Workstation ID number
@WSTYPE	User Workstation Type
@YEAR	Current Year
@YEARDAY	Current Day of Year

ISL System Variable Reference

This section is an A-Z reference of the system variables supported by the ISL. Each system variable includes the following information:

- **Description**: Summarizes the function of the system variable.
- **Type/Size**: Contains the symbol that represents the data type and size of the field or total returned.
- **Syntax**: Provides the proper way to specify the system variable and any arguments, as well as a description of each argument.
- **Remarks**: Gives more detailed information of the system variable, its arguments, and how the system variable is used.
- **Example**: Includes an example of the system variable being used in a script. *This section may not appear in the detail description of each system variable.*
- **See Also**: Names related system variables, commands, functions, other documentation to consult, etc.

ACTIVE_LANGID

Description

This system variable holds the ID number of the currently selected language.

Type/Size

N9

Syntax

@ACTIVE_LANGID

Remarks

- This system variable is Read-Only.
- This system variable is only available on SAR Ops.

ADDXFER_CHK_FROM

Description

This system variable returns the check number of the check being transferred from when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the check number of the check being added to the current check.

Type/Size

N9

Syntax

@ADDXFER_CHK_FROM

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_CHK_TO

Description

This system variable returns the check number of the new check being transferred to (if the check number is changed) when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the check number of the check that is receiving the newly added check.

Type/Size

N9

Syntax

@ADDXFER_CHK_TO

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_GRP_FROM

Description

This system variable returns the table group number of the check being transferred from when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the table group number of the check being added to the current check.

Type/Size

N9

Syntax

@ADDXFER_GRP_FROM

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_GRP_TO

Description

This system variable returns the table group number of the new check being transferred to (if the check number is changed) when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the table group number of the check that is receiving the newly added check.

Type/Size

N9

Syntax

@ADDXFER_GRP_TO

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_RVC_FROM

Description

This system variable returns the Revenue Center number of the check being transferred from when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the Revenue Center number of the check being added to the current check.

Type/Size

N3

Syntax

@ADDXFER_RVC_FROM

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_RVC_TO

Description

This system variable returns the Revenue Center number of the new check being transferred to (if the check number is changed) when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the Revenue Center number of the check that is receiving the newly added check.

Type/Size

N3

Syntax

@ADDXFER_RVC_TO

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_TBL_FROM

Description

This system variable returns the table number of the check being transferred from when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the table number of the check being added to the current check.

Type/Size

N9

Syntax

@ADDXFER_TBL_FROM

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ADDXFER_TBL_TO

Description

This system variable returns the table number of the new check being transferred to (if the check number is changed) when called inside an XFER_CHECK event.

Inside the ADD_CHECK event, this system variable will return the table number of the check that is receiving the newly added check.

Type/Size

N9

Syntax

@ADDXFER_TBL_TO

Remarks

- This system variable is Read-Only.
- This system variable is only valid in ADD_CHECK or XFER_CHECK SIM events.

ALPHASCREEN

Description

This system variable contains the number of the default alpha touchscreen defined for a Revenue Center.

Type/Size

N5

Syntax

@ALPHASCREEN

Remarks

This system variable is Read-Only.

AUTOSVC

Description

This system variable contains the sum of all auto service charges posted to the current guest check.

Type/Size

$12

Syntax

@AUTOSVC

Remarks

This system variable is Read-Only.

BEVERAGE_REQD

Description

This system variable determines whether or not POS Operations should prompt the user for beverages in accordance with the *Beverage Control* feature. During "normal transaction processing," POS Operations checks the items in the transaction to determine if user prompting is appropriate.

Since it is not possible for POS Operations to be in this same state when calling the beverage_reqd variable, the following logic is used internally to determine the value of the @beverage_reqd variable. This should help the SIM developer to understand when POS Operations set the variable to true.

```
Set @beverage_reqd = FALSE
If Beverage Control is on in the Revenue Center then
    Get the Number of Beverages on the check
    If Beverages are being controlled by Guest Count then
       If the Guest Count is greater than zero AND there
       are no beverages, OR the Guest Count is greater than the
       number of beverages then
          Set @beverage_reqd = TRUE
       EndIf
    Else if there are no beverages on the check then
       Set @beverage_reqd = TRUE
    EndIf
EndIf
```

Type/Size

N9

Syntax

@BEVERAGE_REQD

Remarks

This system variable is Read-Only.

Example

```
event inq : 1
   Errormessage "Beverage Control state is ", @beverage_reqd
endevent
```

CCDATE

Description

This system variable contains the expiration date of a credit card that has been read from track data by a magnetic card reader, or has been manually entered.

Type/Size

A4

Syntax

@CCDATE

Remarks

- This system variable is Read-Only.
- This system variable must be accessed within a tender/media event procedure that references a credit card tender type.

See Also

@CCNUMBER system variable

CCNUMBER

Description

This system variable contains the account number of a credit card that has been read from track data by a magnetic card reader, or that has been manually entered.

Type/Size

A30

Syntax

@CCNUMBER

Remarks

- This system variable is Read-Only.
- This system variable must be accessed within a tender/media event procedure that references a credit card tender type.

See Also

@CCDATE system variable

CENTER

Description

This system variable contains the column number that is required to center text in an ISL-defined window.

Type/Size

N9

Syntax

@CENTER

Remarks

- This system variable is Read-Only.
- @CENTER evaluates to -1.
- This system variable can be used as the *column* argument when specifying the **Display** command.

Example

The following event procedure centers the text within an ISL-defined window:

```
event inq : 1
   window 4, 40
   display 1, @center, "In this window, all lines have "
   display 2, @center, "been centered to give it "
   display 3, @center, "that professional "
   display 4, @center, "look."
   waitforclear
endevent
```

See Also

Display command

CHANGE

Description

This system variable is the amount of change due for an overtender.

Type/Size

$12

Syntax

@CHANGE

Remarks

- This system variable is Read-Only.
- This system variable is valid under only two conditions:
    - If in the TMED event
    - If the @TTLDUE system variable equals $0.00

See Also

@TTLDUE system variable

CHECKDATA

Description

This system variable returns a string that contains a facsimile of a guest check created by the current transaction.

Type/Size

String; size depends on data

Syntax

@CHECKDATA

Remarks

- This data is Read-Only.
- The string may consist of zero or more lines that are separated by ASCII newlines, including print formatting characters specifying red ink or double-wide characters.
- The **MakeAscii** command can be used to strip out the print formatting characters in the string.
- This variable should only be accessed in a final_tender event.

See Also

MakeAscii command

CHGTIP

Description

This system variable contains the charged tip for the associated tender in a TMED event.

Type/Size

$12

Syntax

@CHGTIP

Remarks

- This system variable is Read-Only.
- Valid only in a TMED event.
- The @CHGTIP amount is included in the @SVC system variable.

CHK

Description

This system variable contains the object number of the Guest Check Printer assigned to the workstation.

Type/Size

N9

Syntax

@CHK

Remarks

- This system variable is Read-Only.
- This system variable can be used as an argument to the **StartPrint** command.

Example

The event procedure below starts a print job at the Guest Check Printer.

```
event inq : 1
   startprint @chk
      printline "this is a line"
   endprint
   if @printstatus = "Y"
      waitforclear "Print successful"
   else
      waitforclear "Print failed"
   endif
```

See Also

- **StartPrint** command
- ISL Printing

CHK_OPEN_TIME

Description

This system variable returns a string containing the date and time that the current guest check was opened.

Type/Size

A17

Syntax

@CHK_OPEN_TIME

Remarks

This system variable is Read-Only.

CHK_OPEN_TIME_T

Description

This system variable returns the date and time that the current guest check was opened seconds since midnight January 1, 1970.

Type/Size

N9

Syntax

@CHK_OPEN_TIME_T

Remarks

This system variable is Read-Only.

CHK_PAYMNT_TTL

Description

This system variable returns the current payment total.

Type/Size

$12

Syntax

@CHK_PAYMNT_TTL

Remarks

This system variable is Read-Only.

CHK_TTL

Description

This system variable returns the current check total.

Type/Size

$12

Syntax

@CHK_TTL

Remarks

This system variable is Read-Only.

CKCSHR

Description

This system variable contains the guest check cashier number.

Type/Size

N9

Syntax

@CKCSHR

Remarks

This system variable is Read-Only.

CKCSHR_NAME

Description

This system variable contains the guest check cashier's check name.

Type/Size

A16

Syntax

@CKCSHR_NAME

Remarks

This system variable is Read-Only.

CKEMP

Description

This system variable contains the number of the Check Employee, the operator who owns the current guest check.

Type/Size

N9

Syntax

@CKEMP

Remarks

This system variable is Read-Only.

Example

The following example is a standard message exchange between Simphony and a PMS:

```
event inq : 1
    var room_num : a4
    input room_num, "Enter Room Number"
    txmsg "charge_inq",@CKEMP,@CKNUM,@TNDTTL,room_num
                                                    // The first field
(charge_inq) is an
                                                    // example of an identifying
string
                                                    // that the POS might use to
process
                                                    // messsage from the POS.
    waitforrxmsg
endevent

event rxmsg : charge_declined                       // This is one of the PMS
response
                                                    // possibilities
    var room_num : a4
    rxmsg room_num
    exitwitherror "Charge for room ", room_num," declined"
endevent
```

CKEMP_CHKNAME

Description

This system variable contains the check employee's check name, the operator who owns the current guest check.

Type/Size

A16

Syntax

@CKEMP_CHKNAME

Remarks

This system variable is Read-Only.

CKEMP_FNAME

Description

This system variable contains the check employee's first name, the operator who owns the current guest check.

Type/Size

A8

Syntax

@CKEMP_FNAME

Remarks

This system variable is Read-Only.

CKEMP_CHKNAME

Description

This system variable contains the check employee's last name, the operator who owns the current guest check.

Type/Size

A16

Syntax

@CKEMP_LNAME

Remarks

This system variable is Read-Only.

CKID

Description

This system variable contains the current guest check ID.

Type/Size

A32

Syntax

@CKID

Remarks

This system variable is Read-Only.

CKNUM

Description

This system variable contains the number assigned to the current guest check.

Type/Size

N9

Syntax

@CKNUM

Remarks

This system variable is Read-Only.

Example

See example of @CKEMP on page 6-41.

CLIENT_ONLINE

Description

This system variable determines if a workstation is online.

Type/Size

N1

Syntax

@CLIENT_ONLINE

Remarks

This system variable is Read-Only.

Example

```
Event Inq : 2
    if @client_online <> 0
       window 1,60
       display 1,2, "@client_online variable value is ",
       @client_online,". SAR Client is online!"
       waitforclear
    else
       window 1,60
       display 1,2, "@client_online variable value is ",
       @client_online,". SAR Client is offline!"
       waitforclear
    endif
EndEvent
```

DAY

Description

This system variable contains the current date.

Type/Size

N2

Syntax

@DAY

Remarks

This system variable is Read-Only.

Example

The following script will construct a *string_variable* containing the current date in the form dd-mm-yy:

```
event inq : 1
   var date : a9

   call get_date_string
endevent
sub get_date_string
   var month_arr[12] : a3
                                          //Listing of all the months
   month_arr[1] = "JAN"
   month_arr[2] = "FEB"
   month_arr[3] = "MAR"
   month_arr[4] = "APR"
   month_arr[5] = "MAY"
   month_arr[6] = "JUN"
   month_arr[7] = "JUL"
   month_arr[8] = "AUG"
   month_arr[9] = "SEP"
   month_arr[10] = "OCT"
   month_arr[11] = "NOV"
   month_arr[12] = "DEC"
   format date as @DAY, "-", month_arr[@MONTH], "-", @YEAR
                                          // i.e., 10-NOV-01
endsub
```

See Also

@MONTH and @YEAR system variables

DBVERSION

Description

The current database version. For example, if a customer wants to take advantage of all elements in a variable-sized array, the customer may have to specify a more recent @DBVERSION value in the SIM script

Type/Size

N5

Syntax

@DBVERSION

Remarks

This system variable is Read-Only.

DETAILSORTED

Description

This system variable contains a “1” value if detail sorting is enabled or a “0” value if sorting is disabled.

Type/Size

N9

Syntax

@DETAILSORTED

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands

DSC

Description

This system variable contains the total amount of discounts applied to the current guest check. This total is the sum of all percentage and amount discounts on the guest check.

Type/Size

$12

Syntax

@DSC

Remarks

This system variable is Read-Only.

DSC_OVERRIDE

Description

When a manual discount is entered, a SIM ‘Discount’ script can decrease the amount of the discount by setting this variable to the desired discount amount.

Type/Size

$12

Syntax

@DSC_OVERRIDE

Remarks

This system variable is Read-Only.

DSCI

Description

This system variable is an array that contains the discount itemizer totals posted to the current guest check.

Type/Size

$12

Syntax

@DSCI[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits of the *expression* are from 1 to 16.
- This variable will return totals posted to the discount itemizer specified by the array index.
- This variable is similar to the @SI variable.

DTL_CAACCTINFO

Description

This system variable is an array containing the credit authorization account information of a detail item on the current guest check.

Type/Size

A20

Syntax

@DTL_CAACCTINFO[*expression*]

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_CABASETTL

Description

This system variable is an array containing the credit authorization base total of a detail item on the current guest check.

Type/Size

$12

Syntax

@DTL_CABASETTL[*expression*]

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_CAEXPDATE

Description

This system variable is an array containing the credit authorization expiration date of a detail item on the current guest check.

Type/Size

A4

Syntax

@DTL_CAEXPDATE[*expression*]

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_CATIPTTL

Description

This system variable is an array containing the credit authorization tip total of a detail item on the current guest check.

Type/Size

$12

Syntax

@DTL_CATIPTTL[*expression*]

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_CATMEDOBJNUM

Description

This system variable is an array containing the credit authorization tender/media object number of a detail item on the current guest check.

Type/Size

N9

Syntax

@DTL_CATMEDOBJNUM[*expression*]

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_DEFSEQ

Description

This system variable contains the definition sequence number of a detail item.

Type/Size

N3

Syntax

@DTL_DEFSEQ[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_DSC_EMPL

Description

This system variable contains the employee number who is getting the employee meal discount for the specified detail entry.

Type/Size

N9

Syntax

@DTL_DSC_EMPL[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_DSCI

Description

This system variable contains the discount itemizer value for the menu item detail class.

Type Size

N9

Syntax

@DTL_DSCI[*expression*]

Remarks

This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_FAMGRP

Description

This system variable is an array containing the family group of a menu item that is listed in the current guest check detail.

Type/Size

N9

Syntax

@DTL_FMGRP[*expression*]

Remarks

- The expression following the system variable is the menu item's detail number.
- The array limits are 1 to @NUMDTLT.
- This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_INDEX

Description

Index of the detail which fired the SIM event; applicable to the following SIM events:

- EMON_MI
- EMON_MI_VOID
- EMON_MI_RETURN
- EMON_DSC
- EMON_DSC_VOID
- EMON_SVC
- EMON_SVC_VOID
- EMON_TNDR
- EMON_TNDR_VOID

Type/Size

N9

Syntax

@DTL_INDEX

Remarks

- The array limits are 1 to @NUMDTLT.
- This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_IS_COND[i]

Description

This system variable is an array that determines if a Guest Check Menu Item is a condiment.

Type/Size

N1

Syntax

@DTL_IS_COND[*expression*]

Remarks

- The array limits are 1 to @NUMDTLT.
- This system variable is Read-Only.

DTL_IS_VOID[i]

Description

This system variable is set to “Y” if this detail item is a void entry. Otherwise, the variable is set to “N.”

Type/Size

N1

Syntax

@DTL_IS_VOID

Remarks

This system variable is Read-Only.

DTL_MAJGRP

Description

This system variable is an array containing the major group of a menu item that is listed in the current guest check detail.

Type/Size

N9

Syntax

@DTL_MAJGRP[*expression*]

Remarks

- The expression following the system variable is the menu item's detail number.
- The array limits are 1 to @NUMDTLT.
- This system variable is Read-Only.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_MLVL

Description

This system variable is an array containing the Main Menu Level (1-8) of a detail item on the current guest check.

Type/Size

N1

Syntax

@DTL_MLVL[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_NAME

Description

This system variable is an array containing the name of a detail item on the current guest check.

Type/Size

A20

Syntax

@DTL_NAME[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.
- The first name of menu items will be returned.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_OBJNUM

Description

This system variable is an array containing the object number of a detail item on the current guest check.

Type/Size

N9

Syntax

@DTL_OBJNUM[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

Example

See the example for @DTL_MLVL on page 6-67.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_PLVL

Description

This system variable is an array containing the price level (1-8) of a menu item on the current guest check.

Type/Size

N1

Syntax

@DTL_PLVL[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_PMSLINK

Description

This system variable is an array containing the PMS link (1-4) assigned to a detail item on the current guest check.

Type/Size

N2

Syntax

@DTL_PMSLINK[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.
- The PMS Link is defined in the *RVC Parameters* module.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_PRICESEQ

Description

This system variable is an array containing the price sequence number (0-64) of a detail item on the current guest check.

Type/Size

N3

Syntax

@DTL_PRICESEQ[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_QTY

Description

This system variable is an array containing the quantity of a detail item on the current guest check.

Type/Size

N5

Syntax

@DTL_QTY[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_SEAT

Description

This system variable is an array containing the object number of the detail item assigned to a seat number.

Type/Size

N5

Syntax

@DTL_SEAT[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_SLSI

Description

This system variable contains the sales itemizer value for the menu item detail class.

Type/Size

N9

Syntax

@DTL_SLSI[*expression*]

Remarks

This system variable is Read-Only.

See also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_SLVL

Description

This system variable is an array containing the Sub Menu Level (1-8) of a detail item on the current guest check.

Type/Size

N1

Syntax

@DTL_SLVL[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

Example

See the example for @DTL_MLVL on page 6-67.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_STATUS

Description

This system variable is an array containing the status of a detail item on the current guest check.

Type/Size

A12

Syntax

@DTL_STATUS[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.
- The value returned is formatted in hexadecimal digits.

See Also

- **UseSortedDetail** and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_SVC_LINK

Description

This system variable is the current detail's stored value card link, as stored in the check detail.

Type/Size

N9

Syntax

@DTL_SVC_LINK[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

- **UseSortedDetail** and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_SVC_TYPE

Description

This system variable is the current detail's stored value card type, as stored in the check detail.

Type/Size

N9

Syntax

@DTL_SVC_TYPE[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

See Also

- **UseSortedDetail** and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_SVCI

Description

This system variable contains the service charge itemizer value for the menu item detail class.

Type/Size

N9

Syntax

@DTL_SVCI[*expression*]

Remarks

This system variable is Read-Only.

See also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_TAXTTL

Description

This system variable returns the total amount of tax for the detail.

Type/Size

$12

Syntax

@DTL_TAXTTL[*expression*]

Remarks

This system variable is Read-Only.

Example

The following script will print out a line for every item whose tax is greater than 1.00.

```
event inq:1
    var count:N5 = 1
    var i:N5

        if @numdtlt = 0          // if no detail items, quit
            waitforclear "No detail items"
            exitcancel
        endif

        window 10, 50                    // generate window to display at most ten
items
        for i = 1 to @numdtlt            // loop through detail looking for detail
items
            if @dtl_taxttl[i] > 1.00
            display count, 1, "Detail ", @dtl_name[i], " tax total is ",
            @dtl_taxttl[i]
            count = count + 1
        endif

        if count > 10                    // make sure there aren't too many items
            break;
        endif
    endfor
    waitforclear
endevent
```

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_TAXTYPE

Description

This system variable contains the tax types that were active when the corresponding menu item, service charge, or discount detail item was ordered.

Type/Size

A2

Syntax

@DTL_TAXTYPE[*expression*]

Remarks

- This system variable is Read-Write.
- This system variable is represented as a two-digit hex field, ranging from *00* to *FF*. Each bit corresponds to the tax type. For example, *80* corresponds to tax type 1.

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED system variable

DTL_TTL

Description

This system variable is an array containing the total of a detail item on the current guest check.

Type/Size

$12

Syntax

@DTL_TTL[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.

Example

This example is part of a script that checks a current guest check for a certain number of menu items. If four menu items are found, the script will call a subroutine that prints a coupon (`call print_coupon`) and a subroutine that determines how many items are on the check (`call check_grill_list( objnum)`); these subroutine scripts are not shown.

```
event tmed : *
    var dtl_cnt : n3
    var num_grill_items : n3 = 6
    var grill_item[ num_grill_items ] : n5
    var grill_hit : n3
    var objnum : n5

    grill_item[ 1 ] = 1501
    grill_item[ 2 ] = 1510
    grill_item[ 3 ] = 1520
    grill_item[ 4 ] = 1530
    grill_item[ 5 ] = 1540
    grill_item[ 6 ] = 1550

// look through the check, have we ordered 4 grill items
    for dtl_cnt = 1 to @numdtlt
    if @dtl_type[ dtl_cnt ] = "M" AND @dtl_ttl[ dtl_cnt ] > 0
        objnum = @dtl_objnum[ dtl_cnt ]
        call check_grill_list( objnum)
    endif
    if grill_hit >= 4
        call print_coupon
    endif
endevent
```

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_TYPE

Description

This system variable is an array containing the detail type of an item on the current guest check.

Type/Size

A1

Syntax

@DTL_TYPE[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are 1 to @NUMDTLT.
- The detail type will be one of the following:

Type	Description
I	Check Information Detail
M	Menu Item
D	Discount
S	Service Charge
T	Tender/Media
R	Reference Number
C	CA Detail

Example

See the example for @DTL_MLVL on page 6-67.

See also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @NUMDTLT system variables

DTL_TYPEDEF

Description

This system variable returns the detail item type definition for discounts (D), menu items (M), service charges (S), and tenders (T).

Type/Size

Size depends on the detail type:

Detail Type	Size
Discount	A6
Menu Item	A12
Service Charge	A12
Tender	A22
CA	A2

Syntax

@DTL_TYPEDEF[*expression*]

Description

- This system variable is Read-Only.
- The type definition is returned as a hex string. If the discount type definition is E78D, then @dtl_typedef[ ] for that discount will be “E78D,” or an A4.
- For menu items, this variable returns the type definition field from the revenue center (RVC) level menu item class module associated with that menu item.
- For discounts, service charges, and tender media detail items, this variable returns the type definition field from the property level modules.
- For CA detail, this variable returns the type definition from the Check Detail.

Example

This piece of ISL code will scan the detail and display any open-priced menu items or open discount items.

```
event inq:1
    var i:N5
    for i = 1 to @numdtlt                                          // loop through
all detail
        if @dtl_type[i] = "M" AND bit( @dtl_typedef[i], 1 ) <> 0 // check if M and
            bit 1 are set in the M class typedef
        waitforclear "Item ", i, " is open priced MI"              // check if D and
            bit 1 are set in the D typedef
        elseif @dtl_type[i] = "D" AND bit( @dtl_typedef[i], 1 ) <> 0
        waitforclear "Item ", i, " is open DSC"
        endif
    endfor
endevent
```

See Also

UseSortedDetail and **UseStdDetail** commands, and @DETAILSORTED and @DTL_TYPE system variables

DWOFF

Description

This system variable returns printed text to single-wide characters (default) if the @DWON system variable was used to switch text to double-wide characters.

Type/Size

A1

Syntax

@DWOFF

Remarks

- This system variable is Read-Only.
- @DWON is also known as a print directive and can be an argument of the **Printline** command.
- All new lines of text print as single-wide characters.

Example

The ISL statement below prints “Print line” in double-wide characters and red ink, then turns off these print directives.

```
startprint printer
   printline              "----------------------------"
   printline @dwon, @redon,      "chit"
   printline              "----------------------------"
   @dwoff, @redoff
endprint
```

See Also

- @DWON system variable; **Printline** command
- “ISL Printing”

DWON

Description

This system variable prints the *expression* that follows it in double-wide characters.

Type/Size

A1

Syntax

@DWON

Remarks

- This system variable is Read-Only.
- @DWON is also known as a print directive and can be an argument of the **Printline** command.
- Double- and single-wide characters may be mixed on the same line.

Example

The ISL statement below will print “Print line” in double-wide characters and red ink.

```
Printline @dwon, @redon, “Print line”        //prints double-wide in red
ink
```

See Also

- @DWON system variable; **Printline** command
- “ISL Printing”

EMPLDISCOUNT

Description

In a discount event, this variable is the number of the employee discount.

Type/Size

N9

Syntax

@EMPLDISCOUNT

Remarks

- This system variable is Read-Only.

EMPLDISCOUNTEMPL

Description

In a discount event, this variable is the employee number of the discount receiving the employee discount.

Type/Size

N9

Syntax

@EMPLDISCOUNTEMPL

Remarks

- This system variable is Read-Only.

EMPLOPT

Description

This system variable is an array containing the setting of SIM Employee Options #1 through #8, which are defined for the employee who initiated the event.

Type/Size

N1

Syntax

@EMPLOPT[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are from 1 to 8.
- The value returned by the system variable will be the setting of the privilege option code: “0” for OFF, or “1” for ON.
- These values correspond to **ISL Employee Options #1 - #8** in *Employee Classes | Privileges*.
- This system variable can be used to control access to specific events in scripts. For example, to prevent certain employees from initiating a particular event, disable one of the eight available privilege option codes. Within the event, include an ISL statement in which the setting of the corresponding privilege option code is checked. Thus, if the setting is disabled, for example, at this point in the script, an error message is issued, or the employee is directed to take some other action instead of performing the task.

EPOCH

Description

This system variable contains the number of seconds that have expired since midnight January 1, 1970, the EPOCH Time.

Type/Size

N9

Syntax

@EPOCH

Remarks

This system variable is Read-Only.

EVENTID

Description

This system variable is the string that represents the ID of the event being raised. The text is the same as the second parameter in an EVENT statement.

Type/Size

A32

Syntax

@EVENTID

EVENTTYPE

Description

This system variable is the string that represents the type of event being raised (an inquire event is “INQ”). The text is the same as the first parameter in an EVENT statement.

Type/Size

A32

Syntax

@EVENTTYPE

FIELDSTATUS

Description

This system variable contains the Input Status Flag, which is set automatically by the ISL after any of the **WindowEdit** or **WindowInput** command is issued in an event procedure.

Type/Size

A1

Syntax

@FIELDSTATUS

Remarks

- This system variable is Read-Only.
- The Input Status Flag will be either of the following settings:

Flag	Description
Y	indicates that all fields were entered by the operator
N	indicates that some, not all, of the fields were entered by the operator

- This system variable will be set to “Y” if each **DisplayInput** variable has been entered using the **WindowEdit** or **WindowInput** command; otherwise the system variable will be set to “N”.

Example

In the example below, three variables have been defined for the **DisplayInput** command, and all have been set by the user; consequently, @FIELDSTATUS is set to “Y”. If the user entered data for only two of the three fields, @FIELDSTATUS would be set to “N”. Thus, accessing this system variable is most logical after issuing either the **WindowEdit** or **WindowInput** command.

```
event inq : 1
   var data[3] : a20
      window 3, 40
      displayinput 1, 1, data[1], “Enter data 1”
      displayinput 2, 1, data[2], “Enter data 2”
      displayinput 3, 1, data[3], “Enter data 3”
      windowinput
      windowclear
      if @fieldstatus = “Y”
         display 2, @center, “All fields entered”
      else
         display 2, @center, “Some fields not entered”
      endif
      waitforclear
   endevent
```

See Also

WindowEdit[**WithSave**] and **WindowInput**[**WithSave**] commands

FILE_BFRSIZE

Description

This system variable is a user-definable variable that the ISL sets when it expects to read lines greater than 2048 bytes in an open file.

Type/Size

N9

Syntax

@FILE_BFRSIZE

Remarks

This user-definable variable has Read-Write attributes.

Example

If the script is reading lines from a file which is 4K in length, for example, then the script should execute the following line:

```
@FILE_BFRSIZE = 4096
```

See Also

“ISL File Input/Output Commands” on page 7-3

FILE_ERRNO

Description

This system variable is where a Standard Error Number value is saved after every file input/output operation initiated during an event procedure.

Type/Size

N6

Syntax

@FILE_ERRNO

Remarks

This system variable has Read-Write attributes.

- The value will either be 0 or non-zero: 0 means no error occurred, and non-zero indicates an error has occurred. The following table contains the more common non-zero error code values that may be returned by the ISL File I/O commands:

Error Name	Error Number Value	Description
EACCES	13	Permission denied
EAGAIN	11	No more processes
EDEADLK	45	Deadlock condition
EFBIG	27	File too large
EIO	5	I/O error
EISDIR	21	Is a directory
ENOLINK	67	The link has been saved
ENXIO	6	No such device or address
EROFS	30	Read only file system
ESPIPE	29	Illegal seek

The following table lists the possible errors that the File I/O commands may receive:

File Commands	Error Names
FOpen	EACCES, EAGAIN, EISDIR, ENXIO, and EROFS
FClose	ENOLINK
FLock	EACCES, EDEADLK, and ENOLINK
FRead, FReadBfr, and FReadLn	EIO and ENOLINK
FSeek	ESPIPE
FUnlock	EACCES and ENOLINK
FWrite, FWriteBfr, and FWriteLn	EFBIG and ENOLINK

- The @FILE_ERRSTR system variable contains the error message text corresponding to the Error Code. This system variable can be used to display that text if the error occurs.

See Also

- @FILE_ERRSTR system variable
- “ISL File Input/Output Commands” on page 7-3

FILE_ERRSTR

Description

This system variable returns a string containing the Standard Error that occurred during a file input/output operation. The string corresponds to the error code saved in the @FILE_ERRNO system variable.

Type/Size

A80

Syntax

@FILE_ERRSTR

Remarks

- This system variable is Read-Only.
- This string can be used to display the actual error message text, based on the number value saved in the @FILE_ERRNO system variable. Displaying this error message can make it easier to troubleshoot problems with file I/O operations and to verify whether the script was successful in executing a file I/O operation. For example, assume an attempt is made to write to a file with the **FWrite** command and the error code 5 is saved in @FILE_ERRNO. If this file I/O operation is unsuccessful, specifying the @FILE_ERRSTR system variable will allow the string "I/O error" to be displayed.
- To determine the string that will be displayed, refer to the table on page 6-99.

Example

The following script opens a file. If the operation is unsuccessful, an error message will display the cause of the error.

```
event inq : 1
   var fn : N5
   fopen fn, "myfile.dat", read
   if fn = 0
      errormessage @FILE_ERRSTR
      exitcontinue
   endif
endevent
```

See Also

@FILE_ERRNO system variable

FILE_SEPARATOR

Description

This system variable stores the user-defined field separator to be used in all file input/ouput operations.

Type/Size

A1

Syntax

@FILE_SEPARATOR

Remarks

- This system variable has Read-Write attributes.
- In normal ISL File I/O operations, ISL assumes the comma (,) character is the field separator. But if a different field separator is needed, the script must change the @FILE_SEPARATOR system variable.
- If a string with more than one character is assigned to the variable, then only the first character will be used.
- When @FILE_SEPARATOR changes the field separator, all subsequent field operations will use the new field separator until the @FILE_SEPARATOR variable is changed.

FILTER_ACTIVE

Description

This system variable is set to “Y” if seat filtering is active. Otherwise, the variable is set to “N.”

Type/Size

A1

Syntax

@FILTER_ACTIVE

Remarks

This system variable is Read-Only.

FILTER_MASK

Description

This system variable is the current seat filter mask.

Type/Size

A8

Syntax

@FILTER_MASK

Remarks

This system variable is Read-Only.

GRPNUM

Description

This system variable contains the table ID group number assigned to the current guest check.

Type/Size

N9

Syntax

@GRPNUM

Remarks

This system variable is Read-Only.

GST

Description

This system variable contains the number of guests assigned to the current guest check.

Type/Size

N5

Syntax

@GST

Remarks

This system variable is Read-Only.

See Also

@GSTRMNG system variable

GSTRMNG

Description

This system variable contains the number of guests remaining on the current guest check after it has been prorated.

Type/Size

N5

Syntax

@GSTRMNG

Remarks

- This system variable is Read-Only.
- This system variable must be used within an Event Tmed only.
- This system variable should be used in tandem with the @GSTTHISTENDER system variable and the **Prorate** command to determine the remaining guest count on a prorated guest check. In a PMS environment, the PMS may require that the guest count on a check be prorated. For example, if five guests are on a $100 check, and $60 ($20 * 3 = $60) is tendered, the PMS assumes that three of the guests have settled. A typical PMS posting scenario will include a step for prompting the operator to enter the number of guests for the current posting, to associate the guests with the tender.

  ISL provides the same capability. However, implementing this function via a SIM Interface requires 1) knowing the number of guests remaining on the check, and 2) informing Simphony of the number of guests to associate with a tender. Use @GSTRMNG to get the number of guests remaining, then use this value as a condition for requiring an operator to enter the number of guests during a tendering transaction before posting to the PMS. See the example below.
- When the **Prorate** command is active and a tender/media event occurs, @GSTRMNG will contain the number of guests yet to post. When the first tender is posted, @GSTRMNG will be equal to all of the guests on the check. When posting subsequent tenders, @GSTRMNG will be the number of guests remaining. For example, if there are five guests on a check, @GSTRMNG will be five. But if three are prorated with the first tender, @GSTRMNG will be two upon the second round of proration.

Example

The following subroutine implements guest count proration:

```
sub prorate_guests
    var num_guests : n5
    prorate

    // If there are still guests left on this check to be prorated,
    // ask the user how many guests this check.
if @gstrmng > 0
    forever
        input number_guests, "Number of guests this tender?"
        if number_guests > @gstrmng
                errormessage "Max guests is ", @gstrmng
            else
                break
            endif
        endfor

    // Prorate this many many guests for this tender. Next time around,
    // this many guests will be subtracted from @gstrmng.

        @gstthistender = num_guests
    endif
endsub
```

See Also

- @GST and @GSTTHISTENDER system variables
- **Prorate** command

GSTTHISTENDER

Description

This system variable contains the number of guests on the current guest check associated with a split tender when proration is active.

Type/Size

N5

Syntax

@GSTTHISTENDER

Remarks

- This system variable has Read-Write attributes.
- This system variable must be used within an Event Tmed only.
- Use this system variable in tandem with the @GSTRMNG system variable and **Prorate** command to properly prorate guest count for a PMS via a SIM Interface. When set, @GSTTHISTENDER will define the number of guests that are prorated during a tendering transaction. For explanation, see the detail description of @GSTRMNG.
- If the PMS requires prorated guest counts when posting tenders, the SIM script must set @GSTTHISTENDER.

Example

See example for @GSTRMNG on page 6-106.

See Also

- @GST and @GSTRMNG system variables
- **Prorate** command

GUID

Description

This system variable returns a string array with the GUID of the current check.

Type/Size

A40

Syntax

@GUID

Remarks

- This system variable is Read-Only.

HEADER

Description

This system variable is string array with 48 elements. The @HEADER[] array is unique to each event. This means that each event can begin writing to the array starting at index 1, rather than at the next available index.

Type/Size

A32

Syntax

@HEADER[*expression*]

Remarks

- This system variable is Read-Write.
- This system variable is only used with the **Print_Header** Event. All transaction system variables are still valid in this event. User input is still allowed, as are file operations and display manipulation. See page 7-62.

See Also

Print_Header and **Print_Trailer** events and **@TRAILER** system variable.

HOUR

Description

This system variable contains the current hour of the day.

Type/Size

N2

Syntax

@HOUR

Remarks

- This system variable is Read-Only.
- The value returned will be from 0 to 23.
- The hour will be in Military Time format. For example, 2 pm will be returned as “14.”

IGNORE_PRMT

Description

This system variable must be set to a non-zero value to enable the keyboard macro command to pass the [Enter] key to general operator prompts.

Type/Size

N5

Syntax

@IGNORE_PRMT=*integer*

Remarks

This system variable has Read-Write attributes.

INEDITCLOSEDCHECK

Description

This system variable is set to “1” if this is an edit closed check entry. Otherwise, the variable is set to “0.”

Type/Size

N1

Syntax

@INEDITCLOSEDCHECK

Remarks

This system variable is Read-Only, and is related to the Simphony function “Adjust Closed Check”.

INPUTSTATUS

Description

This system variable sets the User Input Status Flag if the **ContinueOnCancel** command is executed.

Type/Size

N9

Syntax

@INPUTSTATUS

Remarks

- This system variable is Read-Only.
- The User Input Status Flag will be set to one of the following:

Flag	Description
0	indicates that the user canceled any input by pressing [Cancel]
1	indicates that the user entered all valid data

See Also

ContinueOnCancel command

INREOPENCLOSEDCHECK

Description

This system variable is set to “1” if this is reopen closed check entry. Otherwise, the variable is set to “0.”

Type/Size

N1

Syntax

@INREOPENCLOSEDCHECK

Remarks

This system variable is Read-Only.

INSTANDALONEMODE

Description

This system variable determines if the workstation is offline.

Type/Size

N1

Syntax

@INSTANDALONEMODE

Remarks

This system variable is Read-Only.

Example

```
Event Inq : 2
   if @InStandaloneMode <> 0
      window 1,65
      display 1,2, "@InStandaloneMode variable value is ",
      @InStandaloneMode,". SAR Client is offline!"
      waitforclear
   else
      window 1,65
      display 1,2, "@InStandaloneMode variable value is ",
      @InStandaloneMode,". SAR Client is online!"
      waitforclear
   endif
EndEvent
```

ISUNICODE

Description

This system variable is set to “Y” if Unicode characters are supported. Otherwise, the variable is set to “N.”

Type/Size

N1

Syntax

@ISUNICODE

Remarks

This system variable is Read-Only.

KEY_CANCEL

Description

This system variable contains the [Cancel] key.

Type/Size

Key

Syntax

@KEY_CANCEL

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

Example

The script below tests that the [Cancel] key was pressed by using @KEY_CANCEL. The operator is prompted to enter a number between 1 and 9. However, if either the [Clear] or [Cancel] key is pressed instead, the script will terminate.

```
event inq : 4
    var key_pressed : key                  // Hold the function key pressed
    var data : a10                         // Hold the number chosen

    forever
        inputkey key_pressed, data, "Type a Number then Enter, Clear to Exit"
        if key_pressed = @KEY_CLEAR
            exitcontinue
        elseif key_pressed = @KEY_CANCEL
            exitcontinue
        elseif key_pressed = @KEY_ENTER
            if data < 0 and data <=10
                waitforclear "You chose ", data, ". Press Clear."
            else
                errormessage "Choose a number between 1 and 10, then press Enter."
            endif
        endif
    endfor
endevent
```

See Also

InputKey command

KEY_CLEAR

Description

This system variable contains the [Clear] key.

Type/Size

Key

Syntax

@KEY_CLEAR

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

Example

See the example for @KEY_CANCEL on page 6-118.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_DOWN_ARROW

Description

This system variable contains the [Down Arrow] key.

Type/Size

Key

Syntax

@KEY_DOWN_ARROW

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_END

Description

This system variable contains the [End] key.

Type/Size

Key

Syntax

@KEY_END

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_ENTER

Description

This system variable contains the [Enter] key.

Type/Size

Key

Syntax

@KEY_ENTER

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

Example

See the example for @KEY_CANCEL on page 6-118.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_EXIT

Description

This system variable contains the [Exit] key.

Type/Size

Key

Syntax

@KEY_EXIT

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_HOME

Description

This system variable contains the [Home] key.

Type/Size

Key

Syntax

@KEY_HOME

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_LEFT_ARROW

Description

This system variable contains the [Left Arrow] key.

Type/Size

Key

Syntax

@KEY_LEFT_ARROW

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_PAGE_DOWN

Description

This system variable contains the [Page Down] key.

Type/Size

Key

Syntax

@KEY_PAGE_DOWN

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_PAGE_UP

Description

This system variable contains the [Page Up] key.

Type/Size

Key

Syntax

@KEY_PAGE_UP

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_RIGHT_ARROW

Description

This system variable contains the [Right Arrow] key.

Type/Size

Key

Syntax

@KEY_RIGHT_ARROW

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

KEY_UP_ARROW

Description

This system variable contains the [Up Arrow] key.

Type/Size

Key

Syntax

@KEY_UP_ARROW

Remarks

- This system variable is Read-Only.
- This system variable is designed for checking keystrokes from the **InputKey** command and setting keyboard macros with the **LoadKyBdMacro** command.

See Also

InputKey and **LoadKyBdMacro** commands

LANG_ID

Description

This system variable is an array that contains the ID numbers of all defined languages.

Type/Size

N9

Syntax

@LANG_ID

Remarks

- This system variable is Read-Only.
- This system variable is only available on SAR Ops.

LANG_NAME

Description

This system variable is an array that contains the language names for all defined languages. The indexing is the same as @lang_id, therefore @lang_name[1] is the name of the language associated with @lang_id[1].

Type/Size

A20

Syntax

@LANG_NAME

Remarks

- This system variable is Read-Only.
- This system variable is only available on SAR Ops.

LASTCKNUM

Description

This system variable contains the previous check number that was assigned to the current guest check.

Type/Size

N9

Syntax

@LASTCKNUM

Remarks

- This system variable is Read-Only.
- A last check number value of 0 indicates that the check number has not changed.

LINE

Description

This system variable contains the number of the current line in the script that is being executed.

Type/Size

N5

Syntax

@LINE

Remarks

- This system variable is Read-Only.
- Use this system variable as a debugging tool.

See Also

@LINE_EXECUTED system variable

LINE_EXECUTED

Description

This system variable contains the number of lines executed since the script began running.

Type/Size

N5

Syntax

@LINE_EXECUTED

Remarks

- This system variable is Read-Only.
- Use this system variable as a debugging tool.

See Also

@LINE system variable

MAGSTATUS

Description

This system variable contains the Magnetic Card Entry Status Flag. The flag indicates whether data was input by swiping a card through a magnetic card reader.

Type/Size

A1

Syntax

@MAGSTATUS

Remarks

- This system variable is Read-Only.
- The Magnetic Card Entry Status Flag will be either of the following settings:

Flag	Description
Y	indicates that data was input by swiping a card through a magnetic card reader
N	indicates that data was input by means other than a magnetic card reader, such as via keyboard entry

- @MAGSTATUS is best used after issuing an **Input** or **WindowInput/Edit** command.

Example

This event captures credit card information, entered manually from a keyboard or electronically from a magnetic card reader. The script uses the @MAGSTATUS system variable to determine the source from which the information is captured.

```
event : 1
    var cardholder_name : a20
    var account_num : a19
    var expire_date : n4
    var track1_data : a79
    var track2_data : a79

    window 3, 78
    touchscreen 16
    displayMSinput 1, 2, cardholder_name{m2, 2, 1, *}, \
        "Read Credit Card or Enter Guest Name", \
        2, 2, account_num{m2, 1, 1, *}, "Enter Account Number", \
        3, 2, expire_date{m2, 3, 1, 4}, "Enter Expiration Date (YYMM)", \
        0, 0, track1_data{m1, *}, " "
    windowinput
    waitforclear
    window 4, 40
    display 1, 2, "Cardholder: ", cardholder_name

    if @MAGSTATUS = "Y"
        display 2, 2, "As read from credit card."
    else
        display 2, 2, "As entered from keyboard."
    endif
    waitforclear
endevent
```

See Also

DisplayMSInput command

MAXDTLR

Description

This system variable contains the maximum size string required to format the transaction detail held in the @TRDTLR system variable.

Type/Size

N9

Syntax

@MAXDTLR

Remarks

This system variable is Read-Only.

See Also

@DTL_*, @MAXDTLT, and @TRDTLR system variables

MAXDTLT

Description

This system variable contains the maximum size string required to format the transaction detail held in the @TRDTLT system variable.

Type/Size

N9

Syntax

@MAXDTLT

Remarks

This system variable is Read-Only.

See Also

@DTL_*, @MAXDTLR and @TRDTLT system variables

MAX_LINES_TO_RUN

Description

This system variable is a debugging tool that can be set to the maximum number of lines in the script to run.

Type/Size

N5

Syntax

@MAX_LINES_TO_RUN = # *of lines*

Remarks

- This system variable has Read-Write attributes.
- This system variable should be set before the lines in the script that are being debugged.

MINUTE

Description

This system variable contains the current minute of the current hour.

Type/Size

N2

Syntax

@MINUTE

Remarks

- This system variable is Read-Only.
- A value returned will be from 0 to 59.

MONTH

Description

This system variable contains the current month of the current year.

Type/Size

N2

Syntax

@MONTH

Remarks

- This system variable is Read-Only.
- The value returned will be from 1 to 12.

Example

See the example for @DAY on page 6-48.

See Also

@DAY and @YEAR system variables

NUL

Description

This system variable specifies that a binary 0 should be sent when printing binary data to a printer. The *@nul* variable is useful only on the *PrintLine* command.

Type/Size

A2

Syntax

@NUL

Remarks

This system variable is used to escape the NUL character.

NUMOPNCHK

Description

This system variable returns the count of Open Checks per Revenue Center.

Type/Size

N5

Syntax

@NUMOPNCHK

Remarks

This system variable MUST be called first to return data when using any of the other OPNCHK_* System Variables.

Example

```
event inq:1
   var num_openchecks       :N9 = 0
   var opencheck_empowner   :N9
   var opencheck_guid       :A50
   var opencheck_number     :N9
   var opencheck_opentime   :A17
   var opencheck_ordertype  :N5
   var opencheck_total      :A7
   var opencheck_wsowner    :N9
   var count            :N9
   var row              :N9
   var chknum           :N9
   var Line1                :A400
   var Line2                :A200


   // Get open checks count
   // This will generate the open check data in the opnchk_*
variables
   num_openchecks = @numopnchk


   // Create a window to show open checks
   window 12,112, "Open Checks"
```

```
   // Display label
  display 1,1, "Check# | EmpOwner |              Guid
| OrderType"
   display 1,85, "| Total | WSOwner | OpenTime"


   // show all open checks in the window
   count = 0
   row = 1
   while count < num_openchecks
                    opencheck_empowner = @opnchk_empowner[count]
                    opencheck_guid = @opnchk_guid[count]
                    opencheck_number = @opnchk_number[count]
                    opencheck_opentime = @opnchk_opentime[count]
                    opencheck_ordertype = @opnchk_ordertype[count]
                    opencheck_total = @opnchk_total[count]
                    opencheck_wsowner = @opnchk_wsowner[count]
                    count = count + 1
                    row = row + 1
                    format Line1 as opencheck_number,"    |  ",
opencheck_empowner,"   | ", opencheck_guid," | ",
opencheck_ordertype,"          |"
                    format Line2 as " ",opencheck_total, " |  ",
opencheck_wsowner,"     | ",opencheck_opentime
                    display row, 1, Line1
                    display row, 70, Line2
                    // Window shows 12 rows at a time . press enter
to continue . then show next 12 rows by replacing the first 12 rows
                    if row = 12
                       waitforenter
                       row = 1
                    endif
   endwhile


endevent
```

See Also

@OPNCHK_EMPOWNER, @OPNCHK_GUID, @OPNCHK_NUMBER, @OPNCHK_OPENTIME, @OPNCHK_ORDERTYPE, @OPNCHK_TOTAL and @OPNCHK_WSOWNER system variables.

NUMDSC

Description

This system variable contains the number of active discounts posted to the current guest check.

Type/Size

N1

Syntax

@NUMDSC

Remarks

This system variable is Read-Only.

NUMDTLR

Description

This system variable contains the number of transaction detail entries posted during the current service round on the guest check.

Type/Size

N5

Syntax

@NUMDTLR

Remarks

This system variable is Read-Only.

NUMDTLT

Description

This system variable contains the number of transaction detail entries posted to the current guest check.

Type/Size

N5

Syntax

@NUMDTLT

Remarks

- This system variable is Read-Only.
- This system variable is used to provide the maximum array limit for the @DTL_* system variables.

Example

See the example for @DTL_MLVL on page 6-67.

See Also

@DTL_* and @NUMDTLR system variables

NUMERICSCREEN

Description

This system variable contains the default numeric entry touchscreen defined for the Revenue Center.

Type/Size

N5

Syntax

@NUMERICSCREEN

Remarks

This system variable is Read-Only.

NUMLANGS

Description

This system variable holds the number of languages in the @lang_id and @lang_name arrays.

Type/Size

N9

Syntax

@NUMLANGS

Remarks

- This system variable is Read-Only.
- This system variable is only available on SAR Ops.

NUMSI

Description

This system variable contains the number of active sales itemizers defined in the *RVC Descriptors* module.

Type/Size

N9

Syntax

@NUMSI

Remarks

This system variable is Read-Only.

NUMSVC

Description

This system variable contains the number of active service charge itemizers defined for the Revenue Center.

Type/Size

N1

Syntax

@NUMSVC

Remarks

- This system variable is Read-Only.
- This system variable will always returns the value 1.

NUMTAX

Description

This system variable contains the number of active tax rates.

Type/Size

N1

Syntax

@NUMTAX

Remarks

This system variable is Read-Only.

OBJ

Description

This system variable is the object number of the detail item for the event.

Type/Size

N9

Syntax

@OBJ

Remarks

This system variable is only valid in the MI*, DSC*, SVC*, and TNDR* events.

OFFLINE LINK

Description

This system variable is used to link to an offline PMS system. For example when the PMS is down, Ops can query the local guest database for account information and post the transaction offline.

Type/Size

N12

Syntax

@OFFLINE LINK

OPNCHK_EMPOWNER

Description

This system variable contains the Object Number of the employee that owns the Open Check.

Type/Size

N9

Syntax

@OPNCHK_EMPOWNER [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK.

See Also

NUMOPNCHK system variable

OPNCHK_GUID

Description

This system variable contains the Open Check GUID (unique identifier).

Type/Size

A40

Syntax

@OPNCHK_GUID [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK.

See Also

NUMOPNCHK system variable

OPNCHK_NUMBER

Description

This system variable contains the Open Check Number.

Type/Size

N9

Syntax

@OPNCHK_NUMBER [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK

See Also

NUMOPNCHK system variable

OPNCHK_OPENTIME

Description

This system variable contains the Date and Time that the Open Check was begun. This variable will return the Local Time in 24 hour format.

Date and Time format: **MM/dd/yyyy HH:MM:SS**

Type/Size

A17

Syntax

@OPNCHK_OPENTIME [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK

See Also

NUMOPNCHK system variable

OPNCHK_ORDERTYPE

Description

This system variable contains the Open Check Order Type ID.

Type/Size

N9

Syntax

@OPNCHK_ORDERTYPE [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK

See Also

NUMOPNCHK system variable

OPNCHK_TOTAL

Description

This system variable contains the Open Check Total Amount.

Type/Size

$12 (Monetary)

Syntax

@OPNCHK_TOTAL [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK

See Also

NUMOPNCHK system variable

OPNCHK_WSOWNER

Description

This system variable contains the Object Number of the Workstation that currently owns the Open Check

Type/Size

N9

Syntax

@OPNCHK_WSOWNER [expression]

Remarks

- The @NUMOPNCHK system variable must be called first to return data using any of the available @OPNCHK_* system variables.
- This system variable is Read-Only.
- The array limits of the expression are from 1 to the count return from @NUMOPNCHK

See Also

NUMOPNCHK system variable

ORDERTYPE

Description

This system variable contains the active order type on the current guest check.

Type/Size

N9

Syntax

@ORDERTYPE

Remarks

This system variable is Read-Only.

ORDERTYPE_NAME

Description

This system variable contains the active order type's name.

Type/Size

A16

Syntax

@ORDERTYPE_NAME

Remarks

This system variable is Read-Only.

ORDR

Description

This system variable is an array containing the object number of a Remote Order or Local Backup Printer defined for Simphony.

Type/Size

N9

Syntax

@ORDR[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are from 1 to 15.
- This system variable can be used as an argument to the **StartPrint** command.

Example

The example below starts a print job at a remote order printer.

```
sub print_message
   startprint @ordr1
      printline "============================"
      printline "Message from ", sender_name
      printline "============================"
      for rowcnt = 1 to 136
         if len(kitchen_msg[rowcnt]) > ""
            printline kitchen_msg[rowcnt]
         endif
      endfor
      printline "======= END MESSAGE ========"
   endprint
            endsu
```

See Also

- **StartPrint** command
- "ISL Printing"

OS_PLATFORM

Description

This system variable is the value of the operating system platform

Value	Description
1	Windows® CE
3	Win 32

Type/Size

N1

Syntax

@OS_PLATFORM

Remarks

This system variable is Read-Only.

PICKUPLOAN

Description

This system variable is the value of the pickup or loan amount.

Type/Size

$4

Syntax

@PICKUPLOAN

Remarks

This system variable is only valid in the PICKUP_LOAN event.

PLATFORM

Description

This system variable contains a character string identifying the hardware platform on which the script is running.

Type/Size

A4

Syntax

@PLATFORM

Remarks

- This system variable is Read-Only.
- The string returned is “Simphony.”

PMSBUFFER

Description

This system variable contains a string that points to the entire message received from the third-party system communicating with Simphony.

Type/Size

String; size depends on the data returned from the third-party system (e.g., PMS)

Syntax

@PMSBUFFER

Remarks

- This system variable has Read-Write attributes.
- The size of data in the PMS buffer is formatted as a string, which can be up to 32,768 bytes in length.
- This system variable is a debugging tool. For example, if the PMS message received by Simphony is suspected of being formatted incorrectly, using @PMSBUFFER the message can be displayed in an ISL-defined window as it is being received.
- Issuing @PMSBUFFER is valid only after a message has been received from the third-party system (e.g., PMS).

See Also

@SHOW_PMS_MESSAGES system variable

PMSLINK

Description

This system variable contains the PMS Link defined in the *RVC Parameters* module.

Type/Size

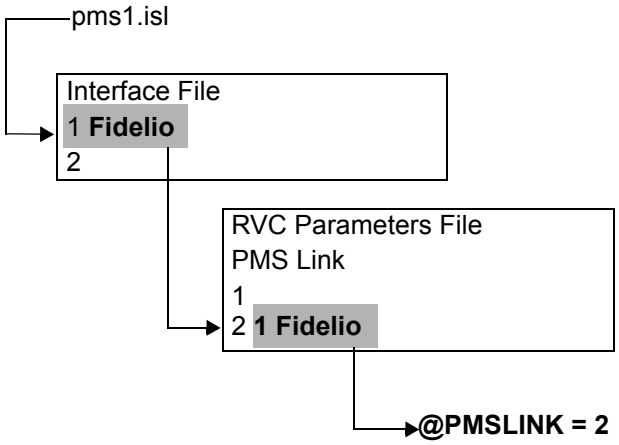
N2

Syntax

@PMSLINK

Remarks

- This system variable is Read-Only.
- The value returned by @PMSLINK will be the PMS defined in the *RVC Parameters* module to which the script is linked. For example, if @PMSLINK is executed by pms1.isl, which is linked to PMS Link #2 "1 Fidelio," then @PMSLINK will be set to "2."



See Also

@PMSNUMBER system variable

PMSNUMBER

Description

This system variable contains the PMS object number, defined in the Interfaces module, to which the script is linked.

Type/Size

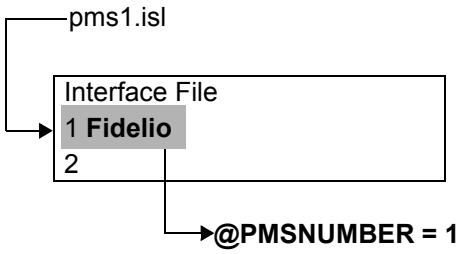
N3

Syntax

@PMSNUMBER

Remarks

- This system variable has Read-Write attributes.
- In order to link the script file to a PMS, this object number is contained in the name of the script file. Thus, if running the script pms1.isl, then @PMSNUMBER will be set to 1.



See Also

@PMSLINK system variable

PREVPAY

Description

This system variable contains the total amount tendered thus far on the current guest check.

Type/Size

$12

Syntax

@PREVPAY

Remarks

This system variable is Read-Only.

PRINTSTATUS

Description

This system variable sets the Print Status Flag to indicate whether a print job has completed successfully or failed.

Type/Size

A1

Syntax

@PRINTSTATUS

Remarks

- This system variable is Read-Only.
- The Print Status Flag will be either of the following settings:

Flag	Description
Y	indicates that a print job completed successfully
N	indicates that a print job failed

Example

The event procedure below uses the setting of @PRINTSTATUS to determine which message to display after issuing the **Printline** command.

```
event inq : 1
   startprint @chk
      printline "this is a line"
endprint
if @printstatus = "Y"
   waitforclear "Print successful"
else
   waitforclear "Print failed"
endif
```

See Also

- **Printline** command
- "ISL Printing"

PROPERTY

Description

This system variable returns the Property Number of the Workstation.

Type/Size

N9

Syntax

@PROPERTY

Remarks

- This system variable is Read-Only.

PRORATETND

Description

This system variable is used in conjunction with the `ProRate` ISL command to calculate prorated values, and only when the `ProRate` command is used outside a SIM Tender Media event.

Type/Size

N9

Syntax

@PRORATETND

Remarks

- The default for this variable is 0 (will not override the tender media record).
- This variable is reset to its default value at the beginning of every event.

See Also

ProRate command

QTY

Description

This system variable is the quantity of the detail item for the event.

Type/Size

N9

Syntax

@QTY

Remarks

This system variable is only valid in the MI*, DSC*, SVC*, and TNDR* events.

RANDOM

Description

This system variable returns a random value between 0 and 2^{32}-1.

Type/Size

N9

Syntax

@RANDOM

RCPT

Description

This system variable contains the object number of the Customer Receipt Printer defined for the System Unit.

Type/Size

N9

Syntax

@RCPT

Remarks

- This system variable is Read-Only.
- This system variable can be used as an argument to the **StartPrint** command.

See Also

- **StartPrint** command
- "ISL Printing"

REDOFF

Description

This system variable contains printed text to black ink (or default ink, e.g., blue).

Type/Size

A1

Syntax

@REDOFF

Remarks

- This system variable is Read-Only.
- @REDOFF is also known as a print directive and can be an argument of the **Printline** command.
- All new lines of text print default ink (i.e., black, blue, etc.).

Note: *The Citizen autocut roll printer does not recognize the first occurrence of this variable after a printline command. The second occurrence, and all succeeded occurrences of this variable, are recognized by the Citizen autocut roll printer.*

This situation does NOT occur with standard MICROS roll printers.

Example

The ISL statement below prints "Print line" in double-wide characters and red ink, then turns off these print directives.

```
startprint printer
   printline        "----------------------------"
   printline @dwon, @redon,      "chit"
   printline        "----------------------------"
   @dwoff, @redoff
endprint
```

See Also

- @REDON system variable; **Printline** command

- "ISL Printing"

REDON

Description

This system variable prints the expression that follows it in red ink.

Type/Size

A1

Syntax

@REDON

Remarks

- This system variable is Read-Only.
- @REDON is also known as a print directive and can be an argument of the **Printline** command.
- Characters in red and black ink can print on the same line.

Note: *The Citizen autocut roll printer does not recognize the first occurrence of this variable after a printline command. The second occurrence, and all succeeded occurrences of this variable, are recognized by the Citizen autocut roll printer.*

This situation does NOT occur with standard MICROS roll printers.

Example

The ISL statement below will print "Print line" in double-wide characters and red ink.

```
Printline @dwon, @redon, "Print line"        //prints double-wide in red
ink
```

See Also

- @REDOFF system variable; **Printline** command

- “ISL Printing”

RETURNSTATUS

Description

This system variable is set to “Y” when the Return and Transaction Return functions are active; otherwise, the variable is set to “N.”

Type/Size

A1

Syntax

@RETURNSTATUS

Remarks

This system variable is Read-Only.

RVC

Description

This system variable contains the number of the Revenue Center to which the script is linked by its Revenue Center PMS Link.

Type/Size

N3

Syntax

@RVC

Remarks

- This system variable is Read-Only.
- This system variable will be set to the object number of the Revenue Center in which the script is running. For instance, if the PMS Link for pms1.isl is defined in Revenue Center #4, then the system variable will be set to 4.

RVC_NAME

Description

This system variable contains the current Revenue Center's name.

Type/Size

A16

Syntax

@RVC_NAME

Remarks

- This system variable is Read-Only.

RVCSERIALNUM

Description

This system variable is an array of two serial numbers that provides an incrementing serial number at the Revenue Center level. This serial number is known as a Revenue Center Sequence Number.

Type/Size

N9

Syntax

@RVCSERIALNUM[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are from 1 to 2.
- Every instance that @RVCSERIALNUM[] is accessed results in its value being incremented by 1, providing a unique Revenue Center Sequence Number each time the system variable is accessed. For example, @RVCSERIALNUM[1] provides one sequence number, and @RVCSERIALNUM[2] provides a second sequence number.
- The sequence numbers returned by the array are independent of each other; consequently, the operation of one sequence number does not affect the other.

Example

The script below will access the sequence number three times, and three unique sequence numbers will be displayed in the ISL-defined window.

```
event inq : 1
   window 3, 40
   display 1, 1, @rvcserialnum[1]
   display 2, 1, @rvcserialnum[1]
   display 3, 1, @rvcserialnum[1]
   waitforclear
endevent
```

See Also

@GUINUM, @GUINUMRVC, and @SYSSERIALNUM[] system variables

RXMSG

Description

This system variable contains the Event ID assigned to the response message sent by a third-party system to Simphony.

Type/Size

A32

Syntax

@RXMSG

Remarks

- This system variable is Read-Only.
- This command is not available on SAR Ops.
- When either the **WaitForRxMsg** or **GetRxMsg** commands are executed, the ISL waits for a return event. The first field of the *Application_Data* segment of the response message is assumed to be the Event ID of the return event. The @RXMSG system variable contains that Event ID.

Example

In the event below, the @RXMSG system variable is used to verify that the Event ID in the returned message is the correct one.

```
event inq : 1
    txmsg "ver_req"                  // Transmit string requesting
                                     //      version of system software
    getrxmsg                         // Wait for response

    if @rxmsg = "ver_rsp"
        rxmsg version_s              // Format message received
        waitforclear version_s       // Display it
    elseif @rxmsg = "ver_err"
        errormessage "Version number invalid"
        exitcancel
    endif
endevent
```

See Also

- **GetRxMsg** and **WaitforRxMsg** commands
- “Application_Data” on page 2-5

SEAT

Description

This system variable contains the number of the active seat on the current guest check.

Type/Size

N5

Syntax

@SEAT

Remarks

This system variable is Read-Only.

SECOND

Description

This system variable contains the current second of the current minute.

Type/Size

N2

Syntax

@SECOND

Remarks

- This system variable is Read-Only.
- A value from 0 to 59 is valid.

See Also

@MINUTE system variable

SHOW_PMS_MESSAGES

Description

This system variable, when set to a non-zero value, outputs all PMS messages to the 8700d.log file, a file that contains debugging messages from all of the Simphony processes.

Type/Size

N5

Syntax

@SHOW_PMS_MESSAGES = *integer*

Remarks

- This system variable has Read-Write attributes.
- @SHOW_PMS_MESSAGES can be set to any integer. However, setting the system variable to 0, will disable its function.
- This system variable can be placed anywhere in a script.
- Using @SHOW_PMS_MESSAGES is another method of debugging PMS messages received by Simphony. @PMSBUFFER is also useful for debugging PMS messages; this system variable contains the actual PMS message received by Simphony.

See Also

@PMSBUFFER system variable

SI

Description

This system variable is an array containing the sales itemizer totals posted to the current guest check.

Type/Size

$12

Syntax

@SI[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits of the *expression* are from 1 to 16.
- This system variable will return the totals posted to the sales itemizer specified by the array index. For example, if the array index references @SI[1], any totals posted to Sales Itemizer #1 will be returned.

SIGCAPDATA

Description

This system variable returns a string that contains the PNG (Portable Network Graphics) data, base64 encoded, of a capture signature image from a Mobile MICROS hand-held terminal.

If used on a WinStation or SAR client, the variable will return `No SIGCAP DATA`.

Type/Size

String; size depends on data

Syntax

@SIGCAPDATA

Remarks

- This variable is read-only.
- This variable should only be accessed in a tender event.
- This variable should only be used in conjunction with the **TxMsg** command.

Example

```
event tmed : 1
   var room_num : a4
   input room_num, "Enter Room Number"
   txmsg "charge",@CHKEMP,@CHKNUM,@TNDTTL,room_num,@SIGCAPDATA
   waitforrxmsg

endevent
```

SIMDBLINK

Description

This system variable links to the SIMDB DLL to the database. For example, if a property has a PMS System which has two connections—one for live postings and another for room updates to the SIMDB DLL—the two systems can be linked with the @SIMBLINK system variable in SIM

Type/Size

N12

Syntax

@SIMDBLINK

SRVPRD

Description

This system variable contains the active serving period.

Type/Size

N9

Syntax

@SRVPRD

Remarks

This system variable is Read-Only.

STRICT_ARGS

Description

This system variable, when set to a non-zero value, will check whether the minimum or maximum number of variables is specified if the **RxMsg**, **Fread**, **Split**, and **SplitQ** commands are used.

Type/Size

N9

Syntax

@STRICT_ARGS

Remarks

This system variable has Read-Write attributes.

See Also

Fread, **RxMsg**, **Split**, and **SplitQ** commands

SVC

Description

This system variable contains the total amount of service charges posted to the current guest check.

Type/Size

$12

Syntax

@SVC

Remarks

- This system variable is Read-Only.
- The @CHGTIP amount is included in the @SVC system variable.

SVCI

Description

This system variable is an array that contains the service charge itemizer totals posted to the current guest check.

Type/Size

$12

Syntax

@SVCI[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits of the *expression* are from 1 to 16.
- This variable will return totals posted to the service charge itemizer specified by the array index.
- This variable is similar to the @SI variable.

SYSSERIALNUM

Description

This system variable is an array of two serial numbers that provides an incrementing serial number at the property level. This serial number is known as a System Sequence Number.

Type/Size

N9

Syntax

@SYSSERIALNUM[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are from 1 to 2.
- Every instance that @SYSSERIALNUM[] is accessed results in its value being incremented by 1, providing a unique System Sequence Number each time the system variable is accessed. For example, @SYSSERIALNUM[1] provides one sequence number, and @SYSSERIALNUM[2] provides a second sequence number.
- The sequence numbers returned by the array are independent of each other; consequently, the operation of one sequence number does not affect the other.

Example

The script below will access the sequence number three times, and three unique sequence numbers will be displayed in the ISL-defined window.

```
event inq : 1
   window 3, 40
   display 1, 1, @sysserialnum[1]
   display 2, 1, @sysserialnum[1]
   display 3, 1, @sysserialnum[1]
   waitforclear
endevent
```

See Also

@GUINUM, @GUINUMRVC, and @RVCSERIALNUM[] system variables

SYSTEM_STATUS

Description

This system variable contains the shell return status after the **System** command is executed.

Type/Size

N6

Syntax

@SYSTEM_STATUS

Remarks

This system variable is Read-Only.

TAX

Description

This system variable is an array containing the totals posted to the active tax rate on the current guest check.

Type/Size

$12

Syntax

@TAX[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are from 1 to 8.

See Also

@TAXRATE[] system variable

TAXRATE

Description

This system variable is an array containing the tax rate defined for the specified *Taxes* module.

Type/Size

A6

Syntax

@TAXRATE[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits for the *expression* are from 1 to 8.
- The value returned is a string instead of an amount, since the percentage may be any number of decimal digits (i.e., 5.1265).

See Also

@TAX[] system variable

TAXVAT

Description

This system variable returns the Value Added Tax amount for Tax Rate “X”.

Type/Size

$12

Syntax

@TAXVAT[*expression*]

Remarks

- This system variable is Read-Only.

See Also

@TAX[] system variable

TBLID

Description

This system variable contains the sequence number of the table ID assigned to the current guest check.

Type/Size

A4

Syntax

@TBLID

Remarks

This system variable is Read-Only.

See Also

@TBLNUM system variable

TBLNUM

Description

This system variable contains the sequence number of the table ID assigned to the current guest check.

Type/Size

N9

Syntax

@TBLNUM

Remarks

This system variable is Read-Only.

Example

This event is used to send to the PMS a message indicating which table has just paid its check.

```
event final_tender

   txmsg "CHECK_PAID", @cknum, @tblnum

// Here we do a 'getrxmsg' to receive the message to fulfill the
// requirements of the protocol, but do not process any data
// associated with the message.

   getrxmsg

endevent
```

TMDNUM

Description

This system variable contains the number assigned to the tender/media associated with this posting.

Type/Size

N9

Syntax

@TMDNUM

Remarks

This system variable is Read-Only.

TNDTTL

Description

This system variable contains the total for this posting, which can be reduced.

Type/Size

$12

Syntax

@TNDTTL

Remarks

- This system variable has Read-Write attributes.
- This system variable must be accessed within a tender/media event.
- The purpose of this system variable is to allow the total due on a check to be updated if necessary. This system variable is best used in an environment where some form of credit limit is applied to purchases, such as in a student meal plan or a frequent diner program for hotel patrons. In a student meal plan, students may have a set amount of credit applied to each meal. For example, assume that each student meal credit limit is $4.50 per meal. If a student surpasses this amount with a purchase of $6.00, the @TNDTTL ($6.00) can be overwritten with the credit limit of $4.50. Then the student can cover the difference of $1.50 with cash, for example.

Example

In the example that follows, the tender total is reduced from $6.00 to $4.50, the allowable student meal credit.

```
event tmed : 1
    txmsg "POST", @tndttl          // send over $6.00
    waitforrxmsg
endevent

event rxmsg: POST
    var new_tnddtl : $10
    rxmsg new_tnddtl               // receive $4.50
    if new_tnddtl > 0
        @tndttl = new_tnddtl
        exitcontinue
    else
        exitcancel                 // cancel
endevent
```

TRACE

Description

This system variable must be set to a non-zero value in order to output each executed ISL statement to the 8700d.log file.

Type/Size

N5

Syntax

@TRACE = *integer*

Remarks

- This system variable has Read-Write attributes.
- The primary usage of this feature is debugging.

TRAILER

Description

This system variable is string array with 32 elements. The @TRAILER[] array is unique to each event. This means that each event can begin writing to the array starting at index 1, rather than at the next available index.

Type/Size

A32

Syntax

@TRAILER[*expression*]

Remarks

- This system variable is Read-Write.
- This system variable is only used with the **Print_Header** event. All transaction system variables are still valid in this event. User input is still allowed, as are file operations and display manipulation. See page 7-62.

See Also

Print_Header and **Print_Trailer** events and **@HEADER** system variable.

TRAININGMODE

Description

This system variable contains the Training Mode Status of an employee.

Type/Size

N1

Syntax

@TRAININGMODE

Remarks

- This system variable is Read-Only.
- The Training Mode Status will be either of the following settings:

Status	Description
Zero	indicates that the employee is not in training mode
Non-zero	indicates that the employee is in training mode

TRCSHR

Description

This system variable contains the Transaction Cashier number of the current guest check.

Type/Size

N9

Syntax

@TRCSHR

Remarks

This system variable is Read-Only.

TRDTLR

Description

This system variable contains transaction detail posted to the current guest check during this service round.

Type/Size

Various (see Remarks)

Syntax

@TRDTLR

Remarks

- This system variable is Read-Only.
- The transaction detail information is designed to provide enough detail, to display or print a basic guest check. If more information is required, it should be exported from the appropriate database files with the Simphony SQL module.
- Each transaction detail entry comprises the following fields:

Field	Type and Size	Description
Detail Type	A1	Indicates type of detail entry: I = Check Information Detail M = Menu Item D = Discount S = Service Charge T = Tender/Media R = Reference Number
Status	A6	Check Detail Status Flag
Number	N9	Object number; set to 0 for reference numbers and check information detail.
Quantity	N5	Quantity; set to 0 for reference numbers and check information detail.

Field	Type and Size	Description
Main Sales Level	N1	Main sales level (between 1 and 8); set to 0 for reference numbers and check information detail.
Sub Sales Level	N1	Sub sales level (between 1 and 8); set to 0 for reference numbers and check information detail.
Total	$12	Total; set to 0 for reference numbers and check information detail.
Name	A22	Contains the detail's name (menu item, discount, etc.). For reference numbers; this contains the actual reference number.

- The transaction detail will be preceded by a Number of Detail Entries field. This field is of the type and size N3, and indicates how many detail entries follow.

- When partial payments are posted to a PMS using prorated itemizers, all the detail on the guest check will be transferred, not just the detail associated with this partial payment. If selective detail is required, the guest check should be split at the POS prior to being posted to the PMS.

See Also

@DTL_*, @MAXDTLR, and @TRDTLT system variables

TRDTLT

Description

This system variable contains all the transaction detail from the current guest check.

Type/Size

Various (see Remarks)

Syntax

@TRDTLT

Remarks

- This system variable is Read-Only.
- The transaction detail information is designed to provide enough detail, to display or print a basic guest check. If more information is required, it should be exported from the appropriate database files with the Simphony SQL module.
- Each transaction detail entry comprises the fields described in the table on page 6-208.
- The transaction detail will be preceded by a Number of Detail Entries field. This field is of the type and size N3, and indicates how many detail entries follow.
- When partial payments are posted to a PMS using prorated itemizers, all the detail on the guest check will be transferred, not just the detail associated with this partial payment. If selective detail is required, the guest check should be split at the POS prior to being posted to the PMS.

See Also

@MAXDTLT and @TRDTLR system variables

TREMP

Description

This system variable contains the number of the Transaction Employee, the employee posting sales to the current guest check.

Type/Size

N9

Syntax

@TREMP

Remarks

- This system variable is Read-Only.
- Typically, the Transaction Employee is also the Check Employee, or the person who originally began the check. However, managers and cashiers may be privileged to be the Transaction Employee if they must post sales to another employee's check. Depending on the Revenue Center Options enabled, sales totals and tender totals will post to either the Check Employee or the Transaction Employee and to their corresponding Cashier Totals, if there is a link.

See Also

@CKEMP system variable

TREMP_CHKNAME

Description

This system variable contains the Transaction Employee's check name, the employee posting sales to the current guest check.

Type/Size

A16

Syntax

@TREMP_CHKNAME

Remarks

This system variable is Read-Only.

TREMP_FNAME

Description

This system variable contains the Transaction Employee's first name, the employee posting sales to the current guest check.

Type/Size

A8

Syntax

@TREMP_FNAME

Remarks

This system variable is Read-Only.

TREMP_LNAME

Description

This system variable contains the Transaction Employee's last name, the employee posting sales to the current guest check.

Type/Size

A16

Syntax

@TREMP_LNAME

Remarks

This system variable is Read-Only.

TTL

Description

This system variable is the amount of the detail item for the event.

Type/Size

$12

Syntax

@TTL

Remarks

This system variable is only valid in the MI*, DSC*, SVC*, and TNDR* events.

TTLDUE

Description

This system variable contains the total due for the current guest check.

Type/Size

$12

Syntax

@TTLDUE

Remarks

This system variable is Read-Only.

TXBL

Description

This system variable is an array containing the taxable sales itemizer on the current guest check.

Type/Size

$12

Syntax

@TXBL[*expression*]

Remarks

- This system variable is Read-Only.
- The array limits of the *expression* are from 1 to 8.

TXEX_ACTIVE

Description

This system variable checks if the Tax is exempt at the specified level.

Type/Size

N1

Syntax

@TXEX_ACTIVE[*expression*]

Remarks

- This system variable is Read-Only.

USERENTRY

Description

This system variable contains the data entered by an operator prior to pressing the SIM Inquiry key.

Type/Size

A20

Syntax

@USERENTRY

Remarks

- This system variable is Read-Only.
- @USERENTRY will contain the data entered prior to pressing the SIM Inquiry key. For example, if an operator enters “123” then presses the SIM Inquiry key, @USERENTRY will contain “123.”

VALD

Description

This system variable contains the object number of the Validation Chit Printer assigned to the workstation.

Type/Size

N9

Syntax

@VALD

Remarks

- This system variable is Read-Only.
- This system variable can be used as an argument to the **StartPrint** command.

See Also

- **StartPrint** command
- “ISL Printing”

VARUSED

Description

This system variable contains the number of bytes of variable space used by the script at the points where variables are referenced.

Type/Size

N5

Syntax

@VARUSED

Remarks

This system variable is Read-Only.

VERSION

Description

This system variable contains the version designation of the SIM.

Type/Size

Various (see Remarks)

Syntax

@VERSION

Remarks

- This system variable is Read-Only.
- This system variable returns text of varying lengths.

Example

The event below draws a window and displays the SIM version number.

```
event inq : 1
   window 1, 30
   display 1, @center, "The SIM Ver is: ", @version
endevent
```

VOIDSTATUS

Description

This system variable is set to “Y” when the Void and Transaction Void functions are active; otherwise, the variable is set to “N.”

Type/Size

A1

Syntax

@VOIDSTATUS

Remarks

This system variable is Read-Only.

WARNINGS_ARE_FATAL

Description

This system variable interrupts script processing with a fatal error if variable overflow occurs.

Type/Size

N5

Syntax

@WARNINGS_ARE_FATAL

Remarks

- This system variable has Read-Write attributes.
- By default, no error is reported when strings, reals, or integers overflow the variables to which they are assigned; the values are truncated to fit the variables. The @WARNINGS_ARE_FATAL system variable can be set to handle the instance when the script writer wants the ISL to report a fatal error if a variable overflow occurs. If @WARNINGS_ARE_FATAL is set equal to 1, variable overflow will cause a fatal error, thereby interrupting script processing.
- Specify @WARNINGS_ARE_FATAL at the top of the script.

Example

```
@WARNINGS_ARE_FATAL = 1

event tmed : 9
   var room : a6
   var guest_name : a20

window 4, 22, "Room Charge"
.
.
.
```

WCOLS

Description

This system variable contains the number of columns in the ISL-defined window currently displayed.

Type/Size

N9

Syntax

@WCOLS

Remarks

This system variable is Read-Only.

WEEKDAY

Description

This system variable contains the day of the week.

Type/Size

N1

Syntax

@WEEKDAY

Remarks

- This system variable is Read-Only.
- Valid values range from 0 - 6, where 0 is Sunday.

WROWS

Description

This system variable contains the number of rows in the ISL-defined window currently displayed.

Type/Size

N9

Syntax

@WROWS

Remarks

This system variable is Read-Only.

WSID

Description

This system variable contains the workstation ID number.

Type/Size

N9

Syntax

@WSID

Remarks

This system variable is Read-Only.

WSTYPE

Description

This system variable is the User Workstation type, such as SAR Client of Mobile MICROS.

Type/Size

N9

Syntax

@WSTYPE

Remarks

- This system variable is Read-Only.
- The workstation types correspond to the type field in the workstation defintition and are as follows:

  1 = Mobile MICROS

  2 = SAR Client

  3 = KWS4

  4 = POSAPI

YEAR

Description

This system variable is at least a two-digit number that contains the number of years since 1900.

Type/Size

N2 or N3. This will be a two-digit number up until the year 2000, when it becomes a three-digit number.

Syntax

@YEAR

Remarks

This system variable is Read-Only.

Example

The year 1999 would be 99 (i.e., 1999-1900), the year 2000 would be 100 (2000-1900), and the year 2015 would be 115 (2015-1900).

See Also

@DAY and @MONTH system variables

YEARDAY

Description

This system variable contains the number representing the current day of the year.

Type/Size

N3

Syntax

@YEARDAY

Remarks

- This system variable is Read-Only.
- A valid value will be from 0 to 365.

Chapter 7

ISL Commands

In This Chapter

This chapter contains a summary and an A-Z reference of all ISL commands, as well as a discussion of format specifiers used in command syntax.

Commands

The Interface Script Language (ISL) provides commands to display information, get operator entries, display touchscreens, execute keyboard macros, as well as transmit and receive messages over the interface. This chapter contains a detail description of each ISL command.

Command Summary

For quick reference, a summary of commands in alphabetical order and in order by category of function begins on page 7-16.

File I/O Operations

A brief introduction and discussion of file I/O commands and system variables is also included in this chapter. Before attempting any file I/O operations for the first time, review this discussion and the detail descriptions of the applicable file I/O commands.

Format Specifiers

This language element can be part of the syntax of certain commands. Format specifiers can be used to change the format of both input and output data. Review "Using Format Specifiers" on page 7-5 to learn the ways in which this language element can be used in command syntax.

ISL File Input/Output Commands

The ISL interpreter includes commands for file operations similar to those offered by languages such as C and BASIC. Anyone familiar with these languages should be comfortable with the ISL file I/O commands.

All file processing involves the following three steps in the order listed:

- open the file
- perform all read and write operations
- close the file

The FOpen Command

When a file in ISL is opened using the **FOpen** command, it is assigned a file number between 1 and 10. While no other file commands can modify this value, this file number is required with all the ISL File Input/Output commands. Since this value is a normal ISL integer, it can be passed into subroutines. The file number's value, when the **FOpen** command is called, will be ignored.

Since ISL is intended to run in a multiprocessing environment, it also has commands for "locking files." This means that if a script has to read a file, it has the capability to prevent other programs from changing the file while it is being read.

File I/O System Variables

All file operations affect two system variables: @FILE_ERRNO and @FILE_ERRSTR. Programmers will recognize these two variables as corresponding to the C "errno" variable and the C "strerror( )" function. If an error has been detected, then @FILE_ERRNO will be set to a non-zero value, and @FILE_ERRSTR will be the readable string describing the condition.

ISL maintains a temporary internal buffer for reading and writing data to and from a file. This buffer is normally set at 2048 bytes. The size of this buffer is available in the system variable @FILE_BFRSIZE.

If a script's file operations require reading or writing data lines greater than 2K, then the script should change the size of the buffer by directly changing the value of @FILE_BFRSIZE. The file buffer size applies to all files used in the script.

For example, if the script is reading lines from a file which is 4K in length, then it should execute the following line:

```
@FILE_BFRSIZE = 4096
```

Input/Output File Format

In general, ISL file handling is geared for reading and writing ASCII files, specifically, comma-separated files (i.e., the files exported and imported via the Simphony Data Access Service). In this format, integers and real values appear without quotes, and non-numeric values appear within quotes.

For example, an employee file may look like this:

```
134,"Tooher","Daniel",100.00,"12FE"
156,"Collins","Michael",150.00,"12FF"
179,"Blaine","Richard",125.00,"56BB"
```

ISL has commands for automatically breaking these comma-separated fields into variables, and writing variables as comma-separated lines. If the format of each line is not a list of fields, then commands exist to read an individual line into a string, as well as writing an individual string to a file.

Using Format Specifiers

In general, the default behavior for entering data and displaying data in ISL is sufficient for most needs. However, it is necessary sometimes to change the default behavior to suit the application at hand. For example, one might want to allow magnetic card data entry, to pad displayed data with 0s instead of spaces, or to center data within a display area.

A variety of ISL commands can be used to accomplish this type of formatting, using a language element called a format specifier. To know if the command takes a format specifier as an argument, look at the syntax for the command in the ISL Command Reference.

What is a Format Specifier?

A format specifier is text enclosed in braces and appears directly after the variable or constant whose input/output behavior is affected. When defined, a format specifier changes the way that the variable or constant is input or output.

For example, the following command will display the contents of the user variable `guest_name` in the prompt area. The format specifier appears directly after the `guest_name` variable.

```
PROMPT guest_name{ 20” }
```

Note: *The meaning of the data within the braces will be explained later.*

Types of Format Specifiers

There are two types of specifiers: input and output. Input specifiers are placed after input variables in commands that get data from a user: **Display**, **DisplayMSInput**, **Input**, and **InputKey**. Output specifiers are placed after variables and expressions that are being converted to ASCII for outputting data to the screen, printer, or a message to a PMS.

Specifier Attributes

The general layout of a format specifier is:

{ [*input_specifier*] [*output_specifer*] }

- The *input_specifier* and *output_specifier* consists of individual specifiers, which are usually one character.
- Spaces and tabs may be used in a format specifier for clarity. The following two format specifiers are equivalent:

```
{-=08}
{ - = 0 8 }
```

- Input and output specifiers can appear within one format specifier. However, not all of the individual specifiers may have meaning. For example, it is possible to put input specifiers after a variable that is going to be displayed, but since data is not being entered into the variable, the input specifiers are meaningless and will be ignored.

Input Specifiers

The input specifiers only have meaning for commands that receive input from the user. They will be ignored if they appear in commands that only output data (for example, the **Display** command).

All input specifiers ***must*** be placed before any output specifiers. If they are present, they must also be placed in the order listed in the following table:

Input Specifier	Description
-	Data being typed in by the operator should not be echoed back to the display
M*n*	Specify the track number (*n* = 1 or 2) and what data to read from the magnetic card. For use with the **Input**, **InputKey**, **DisplayInput**, and **DisplayMSInput** commands only. The M character is case-insensitive

Input Specifier

The - specifier is used to hide data being entered by the operator. For example, authorization codes or passwords should not be echoed to the display as the operator types them in. The following command prompts the operator for an authorization code, but echoes it back to the display as it is being typed:

```
Input auth_code, "Enter authorization code"
```

It can be rewritten so that no data is echoed:

```
Input auth_code{-}, "Enter authorization code"
```

If the - specifier is used in commands that require both operator input and the data to be displayed, then not only will the data not be echoed, it will also not be displayed in the window after it is entered. Instead, the field will contain asterisks where data is expected.

M Input Specifier

The M specifier is used when magnetic card data may be entered in lieu of the operator typing the data in. The M specifier defines whether the data is on a mag card, and which track and field the data should be read from. For example, it is possible to use the M specifier to get an authorization code from track 2, field 1, starting offset 3, and copy in 10 characters.

There are two M formats:

- Format 1 Syntax: M*n*,*
- Format 2 Syntax: M*n*, *field*, *start*, *count* | *

These fields are defined as follows:

Field	Description
M*n:*	The track number (M1 or M2). This can be followed by a star (*) to specify all fields on the track, or use the remaining fields in this table to read specific information.
field:	The field position within the specified track. This is a positive integer.
start:	The starting offset (character) within the field. For example, if the last four characters of the "Blaine Richard" string needed to be removed, start the offset at 11.
count:	Number of characters to be read from the *start* (first character) to the end of the *field* (place an asterisk * to include all characters).

Format One

In format 1, the data from the entire track (1 or 2) will be placed into the variable when the mag card is swiped. The following command allows the user to enter a code or swipe a magnetic card:

```
Input auth_code{ M2,* }, "Enter authorization code"
```

If the mag card is swiped, then all the data from track 2 (M2) will be placed into the variable `auth_code`.

Format Two

Format 2 defines exactly where the data in the track occurs. If the authorization code appears in field 1 of track 2, and furthermore, starts at character 3 in the field and consists of 10 characters, then the command can be rewritten as:

```
Input auth_code{ M2,1,3,10 }, "Enter authorization code"
```

If the operator swipes the card, the appropriate data will be extracted from the field and placed into `auth_code`.

Note: *A * can be substituted for* count*, to specify ALL data from the start offset in the field.*

Field Positions for Credit Cards

The following is an illustration of the standard *field* positions for credit cards:

Track 1:

Field #	1	2	3
Data	16/19 Digit Account Number	26 Alpha Character Account Name	Y Y M M

Track 2:

Field #	1	2
Data	16/19 Digit Account Number	Y Y M M

The following **Input** command allows the operator to enter the credit card name or swipe the card and have the name transferred from track 1, field 2.

```
Input card_name{ M1,2,1,* }, "Enter cardholder name"
```

The following is an illustration of the standard *field* position for the MICROS Employee Card (Note: this card is Track 2 only):

Field #	1
Data	10 Digit Employee Number

The following **Input** command will get the employee number from the operator or the mag card and will not echo the data as it is being entered:

```
Input empl_num{ - M2,1,1,10 }, "Enter employee number"
```

Using Both Input Specifiers

Both input specifiers may be used. This command uses both the - and the M specifiers:

```
Input auth_code{ - M2,1,3,10 }, "Enter authorization code"
```

Output Specifiers

Output specifiers are used after variables and expressions that are being converted to ASCII. The output specifiers are similar to the C language printf() specifiers. The following table lists some representative commands for each of these output types:

Commands	Output Type
Display, WaitForConfirm, and Window	Screen
PrintLine	Printing
FWrite	File I/O
TxMsg	PMS

Syntax

The proper syntax for using the *output_specifiers* is as follows:

[<|=|>|*] [+] [0] [*size*] [D|X|O|B] [^] ['] [:*format_string*]

Output specifiers ***must*** also be placed in the order listed in the following table:

Output Specifier	Description
<	Left justification; the *size* specifier may be used to specify the size of the field.
=	Center justification; the *size* specifier may be used to specify the size of the field.
>	Right justification; the *size* specifier may be used to specify the size of the field.
*	Trim leading and trailing spaces; the *size* specifier may be used to specify the size of the field.
+	Place the sign at the start of field.
0	Place the sign at the start of field.
size	Where *size* is the number of the characters in the required field. The *size* must be a positive integer or an expression that is a positive integer.
D	Decimal (Default); display numerics in decimal format.
X	Hexadecimal; display numerics in hexadecimal format.
O	Octal; display numerics in octal format.
B	Binary; display numerics in binary format.
^	Place a space on each side of the data to be displayed.

Output Specifier	Description
"	Place double quotes around the data to be displayed.
:*format_string*	Similar to the BASIC language PRINT USING command. All characters will be displayed except for the # character, which will be replaced by characters from the variable or expression preceding the format specifier.

Examples of Specifiers

The following are examples of how Input and Output Specifiers may be used. For complete examples and explanations of the ISL commands, please see the "ISL Command Reference" on page 7-22.

Input Specifier

The following lines would read data from a Credit Card:

```
displaymsinput 1, 2, cardholder_name{m1, 2, 1, *}, "Enter Guest
Name", \
                2, 2, account_num{m1, 1, 1, *}, "Enter Account Number",
\
                3, 2, expiration_date{m1, 3, 1, 4}, "Enter Expiration"
```

Output Specifiers

Justification Specifiers

The justification specifiers <, =, and > are only meaningful when the size of the expression being formatted is greater than the size of the variable itself.

All integers and decimal expressions are right justified, and all string expressions are left justified, by default. The following section gives examples and shows how these specifiers can be used to justify data:

Expression	Output
125 { 8 }	125
125 { <8 }	125
125 { =8 }	125
125 { >8 }	125
"abc" { 8 }	abc
"abc" { =8 }	abc

* Specifier

The * specifier is used when the expression should be displayed with leading and trailing spaces removed.

Expression	Output
“125” {*}	125
“ 125 ” {*}	125
“ word 1 and word 2 ” {*}	word 1 and word 2

+ Specifier

The + specifier is used to override the default behavior of displaying negative numbers with the - sign to the right of the number by causing the - to appear on the left.

Expression	Output
-891	891-
-891 {+}	-891

If the **SetSignOnLeft** command is executed, then the sign will ***always*** appear on the left side of the number. The + specification in this case will be superfluous.

0 Specifier

The 0 specifier is used to pad the data being displayed with ASCII 0s instead of spaces. The 0 specifier is only meaningful if the *size* specifier is also used:

Expression	Output
199 { 0 }	199
199 { 5 }	199
199 { 05 }	199
199 { <05 }	19900

size Specifier

The *size* specifier defines the width of the expression being displayed. If no *size* specifier is present, then the width of the data formatted will be equal to the number of characters in the data.

If the *size* specifier is 8, then 8 characters will be displayed, irrespective of the width of the actual data being displayed. The output data will be padded with spaces (unless the 0 specifier is used) or truncated if the size specifier is less than the length of the data to be displayed.

There are two types of size specifiers: absolute and expression. All size specifiers must evaluate to positive integers. Negative numbers and/or decimal values are not allowed.

Absolute specifiers are an integer value, for example, 5. The size specified must ***not*** begin with a 0, since the 0 will be mistaken for the 0 specifier.

Expression sizes use standard ISL expressions to specify the size. However, the expression must be enclosed in parentheses, or an error will be displayed.

(In the following example, the value of width is assumed to be 3.)

Expression	Output
“fred”	fred
“fred” { 8 }	fred
“fred” { =8 }	fred
“fred” { (width*2) }	fred
“fred” { width+2 }	ERROR: not enclosed in ( )

D,X,O, and B Specifiers

The radix specifiers (D, X, O, and B) determine the numeric base of the integer expression being displayed. They have no meaning for decimal and string data. The default is base 10 (D).

Expression	Output
100	100
100 {H}	64
100 {B}	1100100
100 {08B}	01100100

:format_string Specifier

The *format_string* is the data that follows the colon : specifier. The *format_string* consists of ASCII characters and the # character. *Format_strings* are used when the data displayed should be interspersed with spaces and/or other characters to fit conventional display methods. For example, a 10-digit phone number should be displayed as:

```
(nnn)nnn-nnnn
```

When the SIM encounters a *format_string*, all # characters will be replaced with data from the preceding expression. All other characters will be output as-is. Characters are replaced starting from the ***right*** side of the format string.

Output format specifiers may be used in a format specifier along with the *format_string* specifier. Any # characters in excess of the expression being formatted will be replaced with spaces, unless the 0 output specifier is used.

For example, to display a U.S phone number, assume that the variable `phone_num` contains the phone number and is equal to 3012108000. Also assume that `room_num` contains a room number and is equal to 17031.

Expression	Output
phone_num	4432858000
phone_num {:###-###-####}	443-285-8000
phone_num {:(###) ###-####}	(443) 285-8000
room_num	17031
room_num {:##-###}	17-031
room_num {:Floor ## room ###}	Floor 17 room 031

It may be necessary sometimes to display the # character and not have it replaced with a character from the output expression. In this case, precede the # character with a single quote.

Expression	Output
phone_num {:Phone ‘# ###-###-####}	Phone # 443-285-8000

It is possible to include format specifiers after each expression being formatted in one command. For example:

```
TXMSG room_number { 04 }, guest_name { <24 }, @cknum
```

Using Input and Output Specifiers Together

Input and output specifiers may be used within the same syntax in the **DisplayInput** and **DisplayMSInput** commands only.

Command Summary

For quick reference, this section contains an alphabetical listing and brief description of all ISL commands.

Note: *ISL commands are listed by category in Appendix C.*

REMEMBER, the commands that *require* either the **StartPrint** command or **Window** command in order to operate correctly are listed in the table below with the following designation:

- (P) for StartPrint

  or

- (W) for Window

Command	Description
Beep	Sound the beeper.
Break	Break out of the current 'For' loop.
Call	Call a subroutine procedure.
ClearArray	Clear an array.
ClearChkInfo	Clears check information detail lines in buffer.
ClearIslTs	Clear any previously defined touchscreen keys.
ClearKybdMacro	Clear macro key definitions.
ClearRearArea	Clears the contents of the customer display.
ContinueOnCancel	Continue processing script even if the [Cancel] or [Clear] key is pressed after an Input command has been issued.
Display (W)	Display text or a field at a defined place within a window.
DisplayInput (W)	Display an input field within a window.
DisplayInverse	Display input field in inverse video.
DisplayIslTs	Display an ISL-defined touchscreen.
DisplayKBArea	Display data in the keyboard entry area of a Keyboard Workstation.

Command	Description
DisplayMSInput (W)	Display an input field within a window and allow magnetic card swipe to satisfy field entry.
DisplayRearArea	Display up to 20 characters on the POS workstation customer display.
DisplayTouchscreen	Displays a workstation touchscreen after a SIM event exits.
DLLCall	Calls a function contained in the DLL. Refer to page F-5.
DLLCall_STDCall	Calls a function contained in the DLL using the STDCall convention. Refer to page F-5.
DLLCallW	Calls a function contained in the DLL with Unicode. Refer to page F-5.
DLLFree	Frees a loaded DLL. Refer to page F-5.
DLLLoad	Loads an external DLL. Refer to page F-5.
ErrorBeep	Sound an error beep.
ErrorMessage	Display an error message and continue.
Event...EndEvent	Indicate the start and end of an Event procedure. The following events are supported: Inq Tmed RxMsg Final_Tender Print_Header Print_Trailer
ExitCancel	Exit a script and cancel the current tendering operation.
ExitContinue	Exit a script and continue the current tendering operation.
ExitOnCancel	Exit a script when the [Cancel] or [Clear] key is pressed after an Input command has been issued.
ExitWithError	Display a defined error message and exit the script.
FClose	Close a file.
FGetFile	Gets a file from the SIM file service:
FLock	Lock a file.
FOpen	Open a file.

Command	Description
For...EndFor	Perform commands a specified number of times.
ForEver...EndFor	Perform commands an indefinite number of times.
Format	Concatenate one or more variables into a string.
FormatBuffer	Format a non-printable string into a printable string.
FormatQ	Concatenate one or more variables into a string and enclose the string in quotes.
FormatRaw (P)	This command allows a SIM script to send up to 2 Kilobytes of raw (un-altered) data to only IDN, Serial, IP, and Bluetooth printers.
FPutFile	Puts a file into the SIM file service server
FRead	Split the next line read from a file into the variables specified in the statement.
FReadBfr	Read the number of bytes specified in the command.
FReadLn	Read the entire line into a string variable.
FSeek	Go to a specified position in an open file.
FUnLock	Unlock a locked file.
FWrite	Write to a formatted file.
FWriteBfr	Write a specified number of bytes.
FWriteLn	Write an entire line.
GetEnterOrClear	Wait for the [Enter] or [Clear] key to be pressed.
GetTime	Retrieve current time.
If...Else[If]...EndIf	Execute commands if the specified condition is met.
InfoMessage	Display an informational message and continue.
Input	Capture operator entry for a single field or prompt.
Inputkey	Capture operator entry and a key for a single field or prompt.
LabelFeedToPeel (P)	Feeds printed labels to the label peeling position. Only works on the Epson L90 label printer.
LineFeed (P)	Linefeed one or multiple lines.

Command	Description
ListDisplay (W)	Display a list.
ListInput (W)	Display a list and get an operator selection.
ListInputEx	Display a list and get an operator selection. Does not provide a WROW or WCOL variable.
ListPrint (P)	Print a list.
LoadDbKybdMacro	Load a pre-defined keyboard macro so that it may be executed upon successful completion of a script.
LoadKybdMacro	Load a user-defined keyboard macro so that it may be executed upon successful completion of a script.
LowerCase	Convert a string to lower-case.
MakeAscii	Remove any non-ASCII or non-printable characters from a string.
MakeUnicode	Remove any non-printable characters from a string.
Mid	Set one portion of a string equal to another string.
MSleep	Sleep for the requested number of milliseconds.
PopUpIslTs	Display a touchscreen as a pop-up.
PrintLine (P)	Print specified text and/or fields.
Prompt	Display an operator prompt.
ProRate	Prorate the itemizers for charge posting.
QueueMsg	Hold the PMS message in the Simphony database queue until the PMS is online.
[Retain/Discard]GlobalVar	Retain or discard global variables between transactions.
Return	Return from a subroutine.
ReTxMsg	Retransmit a message.
RxMsg	Define the format of a message received over the interface.
SaveChkInfo	Insert check information detail into the check.

Command	Description
SaveRefInfo	Save information as tender/media reference detail.
SaveRefInfox	Save information as tender/media reference detail with reference type.
ScanBarcode	Used by SIM to Scan Barcodes that contain more than 40 characters (e.g., QR codes).
SetIslTsKey	Define a touchscreen key.
SetReRead	Re-read the ISL script for new or changed ISL scripts.
SetSignOn[Left/Right]	The minus sign will go on the left or right side, respectively, when formatting numbers.
SetString	Replace all or a specific number of characters in a string with a particular character.
SimDB	Used by SIM to send a request to the SIMDB DLL and then receive a response.
Split	Break a string into separate fields.
SplitQ	Break a string into separate fields and enclose the string in quotes.
StartPrint...\ EndPrint[FF/NOFF] (P)	Print information on a specified printer, with or without a form feed.
Sub...EndSub	Indicate the start and end of a subroutine procedure.
System	Execute a Windows command.
Touchscreen	Activate a touchscreen for the duration of this operation.
UpperCase	Convert a string to upper-case.
UseBackupTender	Use backup tender programmed in the Simphony database.
Use[Compat/ISL]Format	Use Simphony-standard or ISL message format.
Use[ISL/STD]TimeOuts	Use ISL time outs or the standard Simphony error messaging when there is no response from the PMS System.
UseSortedDetail	Consolidated detail is accessible.
UseStdDetail	Raw detail is accessible.
UseTMSFormat	Format messages using the TMS message format.

Command	Description
Var	Declare a variable field of specified type that will be used for input and/or used in an interface message.
WaitForClear	Wait for the [Clear] key before continuing. If no prompt text is supplied, “Press Clear to Continue” is the default.
WaitForConfirm	Wait for an operator confirmation. If no prompt text is supplied, “Press Enter to Continue” is the default.
WaitForEnter	Wait for the [Enter] key before continuing. If no prompt text is supplied, “Press Enter to Continue” is the default.
WaitForRxMsg	Wait for an interface message to be received after a TxMsg has been sent. If no prompt text is supplied, “Please Wait--Sending Message” is the default.
While...EndWhile	Execute a loop structure until an expression becomes FALSE.
Window	Create a window of specified size and optionally display a window title.
WindowClear (W)	Clear a display window.
WindowClose (W)	Close the current window.
WindowEdit[WithSave] (W)	Display the current contents of specified variables within a window and allow them to be edited; optionally require the [Save] key to save entries and exit.
WindowInput[WithSave] (W)	Display the specified fields within a window, without the present contents; optionally require the [Save] key to save entries and exit.
WindowScrollDown	Scroll the current window down one line.
WindowScrollUp	Scroll the current window up one line.

ISL Command Reference

This section is an A-Z reference of ISL commands. The information for each command is organized into the following categories:

- **Description**: Summarizes the function of the command.
- **Syntax**: Provides the proper way to specify the command and any arguments, as well as a description of each argument.
- **Remarks**: Gives more detailed information of the command, its arguments, and how the command is used.
- **POS Setup**: Provides any Simphony database programming required to issue the command successfully.
- **Example**: Includes an example of the command being used in a script.
- **See Also**: Names related commands, functions, system variables, other documentation to consult, etc.

Beep

Description

This command can be used to sound the beeper at a workstation. It should be used for operator confirmation or notification. Note that a separate ErrorBeep command is provided to notify the operator of errors.

Note: *The Beep command currently does not cause the workstation to beep in Simphony as the "Enable Error Beeper" option is not available in the Enterprise Management Console (EMC).*

The command remains so that scripts written for legacy MICROS products using the Beep command will still function in Simphony.

Syntax

Beep

See Also

ErrorBeep command

Break

Description

This is used to break out from a **For** or **Forever** loop. This is especially useful when a **ForEver** loop is executed.

Syntax

Break

Remarks

- The **Break** command will only break out of the **For** or **Forever** loop it is currently in. If the loops are nested, then multiple breaks are required:

```
forever
   forever
      break                  //break out of inner loop
   endfor
   break                     //break out of outer loop
endfor
```

- If the ability to break out of a nested **For** is required, then use a subroutine and **Return** out of the loop instead:

```
sub break_out

   forever
      for i = 1 to 10
         for num = i to count
             if...
                            return
             endif
         endfor
      endfor
   endfor
endsub
```

Example

The following script provides an example of how to break out of a **Forever** loop:

```
event inq : 1
    var user_input : N6
    forever
        input user_input, "Enter a number and press [ENTER]"
        if user_input > 0 AND user_input <99999
            break
        else
            errormessage "Value outside valid range"
        endif
    endfor
    errormessage "Well done"
endevent
```

See Also

For and **ForEver** command

Call

Description

This command is used to call a subroutine defined by the **Sub** command.

Syntax

Call *name*

Argument	Description
name	the name of the subroutine defined by the **Sub** command

Remarks

- The subroutine has access to all the local variables within the Event that called the subroutine, and all global variables in the script file, so these variables may be used to pass parameters. In addition, local variables may be declared in the subroutine.

- When a **Call** is made, ISL will start searching for the subroutine from the top of the program. Therefore, if there are two subroutines with the same name, only the first one will ever get called:

```
event inq:1
   call mysub
endevent

sub mysub                          //this one will get called
   .
   .
   .
endsub

sub mysub                          //this one will not
                                   //  because it is
                                   //  preceded by a
                                   //  subroutine of
                                   //  the same name
   .
   .
   .
endsub
```

- Up to 32 calls can be nested within a subroutine. If there are anymore, an error will occur.

```
sub mysub
   call mysub                          //this will occur 32 times
endsub
```

Example

The following script will call a subroutine to build a window:

```
event inq : 1
   var win_string : a40 = "This window was built in a subroutine!"

   call msg_window
   waitforclear
endevent

sub msg_window
   window 1, len(win_string) + 2
   display 1, 2, win_string
endsub
```

See Also

Sub command

ClearArray

Description

This command sets all elements of the specified array equal to zero if the array is numeric, or null if alphanumeric. By default, arrays are initialized in this way when declared.

Syntax

ClearArray *array_variable*

Argument	Description
array_variable	the name of the array to clear, based on the name of a *user_variable*

Example

The following script allows the user to send up to a 13 line message to the kitchen printer. Before actually sending to the printer, it allows the user the opportunity to edit their work. If the user presses clear when prompted, the *array_variable* is cleared and the user can retype a message.

```
event inq : 1
    var kitchen_msg[13]: a20
    var sender_name: a20
    var rowcnt   : n3
    var term_key : key
    var data_entered: a20

    forever
        call get_message
        window 1, 66
        display 1, 2, "PAGE UP=edit, CLEAR=retype, ENTER=send, CANCEL="quit"
        inputkey term_key, data_entered, ""
        if term_key = @KEY_ENTER
            break
        elseif term_key = @KEY_CLEAR
            cleararray kitchen_msg
        elseif term_key = @KEY_CANCEL
            exitcontinue
        endif
    endfor
    call print_message
endevent
sub get_message
    window 14, 22
    displayinput 1, 2, sender_name, "Enter your name"
    for rowcnt = 1 to 13
        displayinput rowcnt + 1, 2, kitchen_msg[rowcnt], "Enter kitchen message"
    endfor
    windoweditwithsave
endsub
```

(continued from previous page)

```
sub print_message
    startprint @ordr1
        printline "============================"
        printline "Message from ", sender_name
        printline "============================"
        for rowcnt = 1 to 136
            if len(kitchen_msg[rowcnt]) > ""
                printline kitchen_msg[rowcnt]
            endif
        endfor
        printline "======= END MESSAGE ========"
    endprint
endsub
```

ClearChkInfo

Description

This command clears any check information detail lines that have not been written to the Guest Check files and are stored in the guest check information buffer. Normally, this command is used if the script added information to the buffer but, at a later time, decides that the information should not be saved in the Guest Check Files.

Syntax

ClearChkInfo

Remarks

- Check information detail is a type of check detail that can be stored in the Guest Check files via the **SaveChkInfo** command. Typically check information detail lines are used to store customer information, such as name and address, so that it can print on a guest check or a remote order device.
- This command is executed upon exiting the script.
- Keep in mind that, like other types of guest check detail, i.e., totals and definitions, guest check information detail lines are only stored in the Guest Check files temporarily and cleared upon closing a guest check.

POS Setup

Refer to the detail description of **SaveChkInfo** for a brief discussion of the usage of check information detail.

Example

The subroutine below requires that the operator input a string five times, then prompts the operator to confirm saving the information. If the operator responds by pressing the [Clear] key, the check information detail is discarded; otherwise, the information is saved.

```
sub get_info
   var string : A20
   var answer : N5
   var i : N5

   for i = 1 to 5
      input string, "Enter string ", i
      savechkinfo
   endfor
   getenterorclear answer, "Save information?"
   if answer = 0
      clearchkinfo
   endif
endsub
```

See Also

- **SaveChkInfo** command

ClearIslTs

Description

This command clears any touchscreen keys that have been defined using the **SetIslTsKey** command.

Syntax

ClearIslTs

Remarks

All previously defined keys are cleared each time a script executes.

After a touchscreen has been displayed, its keys remain defined, thus, MICROS Systems, Inc. recommends using the **ClearIslTs** command to clear previously defined touchscreen keys when building two or more touchscreens in the same event.

Example

The following example is a subroutine (create_ts) that clears previously defined touchscreen keys before calling another subroutine (set_keys), one that will build a new touchscreen.

```
sub create_ts
clearislts                          //Clear out any previously
                                    // defined touchscreen keys
call set_keys                       //Build the keys needed
.
.
.
endsub
```

See Also

DisplayIslTs, **PopUpIslTs**, and **SetIslTsKey** commands

ClearKybdMacro

Description

This command will clear out any macro keys that have been defined by the **LoadKybdMacro** or **LoadDbKybdMacro** commands since the script started.

Syntax

ClearKybdMacro

Remarks

All macro keys are cleared out when the script is started.

Example

For example, this command may be used if the **LoadKybdMacro** command were issued, but the response from the PMS system was incorrect; the **ClearKybdMacro** would be used to clear the macro in preparation for a rebroadcast or transaction cancel.

```
event inq:1
    loadkybdmacro 11:841          //Load PMS 1 Inquiry Key
    txmsg "inq_1_request"
    waitforrxmsg
endevent

event rxmsg : inq_1_reply
    var status : n5
    rxmsg status
    if status = 0
           errormessage "No Response from PMS "
           clearkybdmacro
       else
           waitforclear "Press Enter to Continue "
       endif
endevent
```

See Also

LoadKybdMacro and **LoadDbKybdMacro** command

ClearRearArea

Description

This command will clear the contents of the customer display.

Syntax

ClearRearArea

Example

```
event inq:1
   DisplayRearArea "Hello"
   WaitForClear "Press clear to clear display"
   clearrearea
end event
```

See Also

DisplayRearArea command

ContinueOnCancel

Description

This command will continue processing the script even if the [Cancel] or [Clear] key is pressed after an **Input** command is issued.

Syntax

ContinueOnCancel

Remarks

- In normal operations, when ISL is waiting for user data after an **Input** command is issued (i.e., **Input**, **WindowInput**, **WindowEdit**,...) and the user presses the [Cancel] key or the [Clear] key at the input prompt, the script will terminate. It may be necessary for the script to continue even if the user has cancelled the entry. If the **ContinueOnCancel** command is executed, then the **Input** commands will not terminate the script if the [Cancel] key or the [Clear] key is pressed. Instead, they will return to the line after the **Input** command. The @INPUTSTATUS system variable will be set to 0 if the user cancelled the input, or 1 if valid data was entered.

- If the **ContinueOnCancel** is used, the script should check all **Input** commands to determine if the user cancelled the input or not.

See Also

ExitCancel, **ExitOnCancel**, **Input**, **WindowEdit**, and **WindowInput** commands

Display

Description

This command can be used to display a message in a window.

Syntax

Display *row, column, expression*[{*output_specifier*}] \
[, *expression*[{*output_specifier*}]...]

Argument	Description
row	the integer *expression* specifying the screen *row* within the defined window where the message will be displayed
column	the integer *expression* specifying the screen *column* within the defined window where the message will be displayed
expression	an *expression* to be displayed; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	the integer *expression* specifying the screen *column* within the defined window where the message will be displayed

Remarks

- Since this command provides information about where to locate the text or fields within the window, a **Window** command must have been executed prior to this command
- The **Display** *row* and *column* must fall within the boundaries of the defined window.

- An error will occur if the data to be displayed extends past the end of the window:

```
window 10, 10, "10 columns"
                                        //ERROR!
display 1, 1, "this line is greater than 10 columns"
```

Example

The following script will display a guest room number and name in a window:

```
event rxmsg : room_info
    var room_num : a5
    var guest_name : a20
        rxmsg room_num, guest_name
        window 1, 40
        display 1, 2, "The guest in room ", room_num, " is ", guest_name
    waitforclear
endevent
```

See Also

Window command; **Chr** function

DisplayInput

Description

This command defines an input field within a window. thus, a **Window** command must have been executed prior to this command. In addition, a **WindowEdit** or **WindowInput** must follow it, or the grouping of **DisplayInput** commands to which it belongs.

Syntax

DisplayInput *row, column, input_variable*[{*input/output_specifier*}],\
prompt_expression[, *prompt_expression,...*]

Argument	Description
row	the integer *expression* specifying the screen *row* within the defined window where the *input_variable* will be displayed
column	the integer *expression* specifying the screen *column* within the defined window where the *input_variable* will be displayed
input_variable	an *array_variable* or *user_variable* that allows user input
{*input/output_specifier*}	one or more of the *input* and *output_specifiers* that determine the format of all input and output fields; see full definition on pages 7-6 through 7-10
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what toenter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- The **DisplayInput** *row* and *column* must fall within the boundaries of the defined window.
- The *prompt_expression* is required.
- **DisplayInput** can be used with the **WindowEdit\Input** commands to build a screen of input fields in order to accept input from the user. Navigating among the input fields is achieved with the movement keys: up arrow, down arrow, home, and end. [Enter] can also be used to navigate, which moves the focus to the next field, and [Clear], which moves the focus to the previous field.
- When a **WindowEdit** or **WindowInput** command is executed, each field displayed using the **DisplayInput** command will be edited in turn.
- The **DisplayInput**, **DisplayMSInput**, **Input**, and **InputKey** commands are the only commands that act on both the Input and Output Specifiers.
- The maximum number of window input entries allowed is 64.

```
for i = 1 to 65
   displayinput 1, i, a[i], "Enter ", i
                              // error when i is 65
endfor
```

- If the *input_variable* to be displayed extends past the end of the window, then an error will occur on the **WindowEdit** or **WindowInput** command, and not the **DisplayInput** command.
- **WindowInput** fields can be edited using the in-place keyboard entry editing feature. The following Type 9 (Keypad) keycodes assign commands to specific keys in the keyboard or touchscreen files:
  - #19—Edit
  - #20—Edit Delete
  - #21—Edit Insert Tggl
- A keyboard entry field can be greater than the 40 characters allowed in a displayed entry field.

Example

The following script will allow input of customer information in a window:

```
event inq : 1
   var rowcnt: n3
   var field_name[5] : a15
   var customer_info[5]: a20
   field_name[1] = "Customer name:"
   field_name[2] = "Company:"
   field_name[3] = "Address:"
   field_name[4] = "City:"
   field_name[5] = "Phone:"

   window 5, 36
   for rowcnt = 1 to 5
       display rowcnt, 2,
       field_name[rowcnt]
       displayinput rowcnt, 16, customer_info[rowcnt],\
             "Enter ", field_name[rowcnt]
   endfor
   windowedit
endevent
```

See Also

Window, **WindowEdit**, and **WindowInput** commands

DisplayInverse

Description

This command can be used to display a message in a window in inverse video. Since this command provides information about where to locate the text or fields within the window, a **Window** command must have been executed prior to this command.

Syntax

DisplayInverse *row, column, expression*[{*output_specifier*}] \
[, *expression*[{*output_specifier*}]...]

Argument	Description
row	the integer *expression* specifying the screen *row* within the defined window where the message will be displayed
column	the integer *expression* specifying the screen *column* within the defined window where the message will be displayed
expression	an *expression* to be displayed; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

- The **DisplayInverse** *row* and *column* must fall within the boundaries of the defined window.
- An error will occur if the data to be displayed extends past the end of the window:

```
window 10, 10, "10 columns"
                                    //ERROR!
displayinverse 1, 1, "this line is greater than 10 columns"
```

Example

The following script will display a guest room number and name in a window:

```
event rxmsg : room_info
    var room_num : a5
    var guest_name : a20
        rxmsg room_num, guest_name
        window 1, 40
        displayinverse 1, 2, "The guest in room ", room_num, " is ", guest_name
    waitforclear
endevent
```

See Also

Display and **Window** commands

DisplayIslTs

Description

This command displays a touchscreen defined by the **SetIslTsKey** command.

Syntax

DisplayIslTs

Remarks

- After a touchscreen has been displayed, its keys remain defined until cleared by the **ClearIslTs** command or until the script terminates
- Sixty temporary touchscreen keys are available.

Example

The subroutine below first clears any previously defined touchscreen keys and displays two touchscreen keys, [YES] and [NO], using the **DisplayIslTs** command. This subroutine displays these keys as the operator is issued a prompt by the system and captures the operator's input.

```
sub get_yes_or_no( ref answer, var prompt_s:A38 )
   var keypress : key
   var data : A20

   clearislts
   setisltskey 2, 2, 4, 4, 3, @KEY_ENTER, "YES"
   setisltskey 2, 6, 4, 4, 3, @KEY_CLEAR, "NO"
   displayislts

   inputkey keypress, data, prompt_s
   if keypress = @KEY_ENTER
      answer = 1
   else
      answer = 0
   endif
endsub
```

See Also

ClearIslTs, **PopUpIslTs**, and **SetIslTsKey** commands

DisplayKBArea

Description

This display keyboard area command displays data in the keyboard entry area of a MICROS Keyboard Workstation (KBWS). Since there are limited display capabilities on the KBWS, this command allows the SIM script writer to display more information on the KBWS display than was normally allowed using the current SIM display commands.

Syntax

DisplayKBArea *prompt_expression*

Remarks

The **DisplayKBArea** accepts a set of display data in a format similar to the **Prompt** command's.

Example

The following script will display a line of data, and then wait for the operator to press clear.

```
event inq:1
    var data:N5

    displaykbarea "Enter a number"
    input data, "Press CLEAR to stop"
    displaykbarea "You entered ", data
    waitforclear "Press CLEAR"

endevent
```

DisplayMSInput

Description

This command defines an input field within a window; therefore, a **Window** command must have been executed prior to this command, and a **WindowEdit** or **WindowInput** must follow it. This command defines an input field within a window that may be entered through the keyboard or touchscreen, or by swiping a magnetic card through the magnetic card reader on the workstation.

Syntax

DisplayMSInput *row, column, input_variable*\
[{*input/output_specifier*}], *prompt_expression*[, *row, column,*\
input_variable{*input/output_specifier*}, *prompt_expression,...*]

Argument	Description
row	the integer *expression* specifying the screen *row* within the defined window where the *input_variable* will be displayed
column	the integer *expression* specifying the screen *column* within the defined window where the *input_variable* will be displayed
input_variable	an *array_variable* or *user_variable* that allows user input
{*input/output_specifier*}	one or more of the *input* and *output_specifiers* that determine the format of all input and output fields; see full definition on pages 7-6 through 7-10
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- This command allows the designer to specify the fields that the operator can enter manually, fields that may be entered from a magnetic card swipe, or fields that may be entered in both fashions. In addition, the location and length of the data to be used on the magnetic card stripe may also be defined.

- After the **WindowInput** command is executed, the system variable @MAGSTATUS will be set to Y if the magnetic card was swiped during the **WindowInput**. It will be set to N if a magnetic card was not swiped. To use @MAGSTATUS in this way, use only one **DisplayMSInput** command with each **WindowInput** entry (otherwise, @MAGSTATUS will be undefined). If more than one **DisplayMSInput** command is needed, use the **Len** function to check if the input string is set to zero (see "ISL Functions" for an explanation of the **Len** function).

- The *prompt_expression* is required.

- **DisplayMSInput** can be used with the **WindowEdit\Input** commands to build a screen of input fields in order to accept input from the user. Navigating among the input fields is achieved with the movement keys: up arrow, down arrow, home, and end. [Enter] can also be used to navigate, which moves the focus to the next field, and [Clear], which moves the focus to the previous field.

- The **DisplayInput**, **DisplayMSInput**, **Input**, and **InputKey** commands are the only commands which act on both the Input and Output Specifiers (please see page 7-5 for more information).

- The maximum window input entries allowed is 64.

```
for i = 1 to 65
displayinput 1, i, a[i], "Enter ", i
                          // error when i is 65
endfor
```

- If the *input_variable* to be displayed extends past the end of the window, then an error will occur on the **WindowInput** command, and not the **DisplayInput** command.

- In the case where row and column is 0, the input field (e.g., `cardholder_name`) is considered hidden and will not be displayed; additionally, it can only be satisfied with a magnetic card, which means no keyboard input is allowed.

  ```
  displaymsinput 0, 0, cardholder_name{m1, 2, 1, *},\
     "Enter Guest Name",...
  ```

  There can be more than one hidden field in a **DisplayMSInput** command. In most cases, the input specification for this field will contain magnetic stripe information.

- The *prompt_expression* for all hidden field(s) will be ignored.

Example

The following script will read the information from Track 1 of a credit card:

```
event inq : 1
    var cardholder_name: a26
    var account_num: n19
    var expiration_date: n4
    var track1_data: a79

    window 3, 78
    displaymsinput 1, 2, cardholder_name{m1, 2, 1, *}, "Enter Guest Name", \
                   2, 2, account_num{m1, 1, 1, *}, "Enter Account Number", \
                   3, 2, expiration_date{m1, 3, 1, 4}, "Enter Expiration"
    windowinput
    waitforclear
endevent
```

See Also

Window, **WindowEdit**, and **WindowInput** commands; **Len** function

DisplayRearArea

Description

This command will display up to 20 characters on the POS workstation customer display (rear display).

This command works on 20-character displays only—8-character displays are ignored.

Syntax

DisplayRearArea *expression[{output_specifier}] [, expression[{output_specifier}]...]*

Example

```
event inq:1
   var text:80
   input text, "Enter data"
   displayrearea "Data:", text
end event
```

See Also

ClearRearArea command

DisplayTouchscreen

Description

This command will display a workstation touchscreen after a SIM event exits.

Syntax

DisplayTouchscreen

DLLCall

Description

This command will call a function contained in the DLL.

Syntax

DLLCall *handle, dll_name([parm1 [parm2 [parm3...]]])*

See Also

- **DLLCallW, DLLFree,** and **DLLLoad** commands
- Appendix F—Windows DLL Access

DLLCall_STDCall

Description

This command will call a function contained in the DLL using the STDCall convention.

Syntax

DLLCall_STDCall *handle, dll_name([parm1 [parm2 [parm3...]]])*

See Also

- **DLLCallW, DLLFree,** and **DLLLoad** commands
- Appendix F—Windows DLL Access

DLLCallW

Description

This command will call a function contained in the DLL with Unicode.

Syntax

DLLCallW *handle, dll_name([parm1 [parm2 [parm3...]]])*

See Also

- **DLLCall, DLLFree,** and **DLLLoad** commands
- Appendix F—Windows DLL Access

DLLFree

Description

This command will free a loaded DLL.

Syntax

DLLFree *handle*

See Also

- **DLLCall, DLLCallW,** and **DLLLoad** commands
- Appendix F—Windows DLL Access

DLLLoad

Description

This command will load the external DLL. The `dllload` command needs to be called only once during the lifetime of the SIM script.

Syntax

DLLLoad *handle, name*

Example

```
event inq:1
   var dll_handle:N9
   dllload dll_handle, "myops.dll"
end event
```

See Also

- **DLLCall, DLLCallW,** and **DLLFree** commands
- Appendix F—Windows DLL Access

ErrorBeep

Description

This command can be used to sound the error beeper at the workstation.

Note: *The ErrorBeep command currently does not cause the workstation to beep in Simphony as the "Enable Error Beeper" option is not available in the Enterprise Management Console (EMC).*

The command remains so that scripts written for legacy MICROS products using the ErrorBeep command will still function in Simphony.

Syntax

ErrorBeep

See Also

Beep command

ErrorMessage

Description

This command can be used to display an error message at the workstation when an incorrect entry is made by the operator.

Syntax

ErrorMessage *expression*[{*output_specifier*}][, *expression*\
[{*output_specifier*}]...]

Argument	Description
expression	an *expression* to be displayed; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

The **ErrorMessage** command expects one error line to be displayed. However, the UWS displays two lines. The error line to be displayed is broken up between the two logical lines. If the line is too long to be displayed, it will be truncated.

See Also

InfoMessage

Example

The following script will display a message indicating that an entry is invalid:

```
event inq : 1
    var menu_choice: n3

    window 3, 23
    display 1, 2, "[1] Edit member info"
    display 2, 2, "[2] Add new member"
    display 3, 2, "[3] Exit"
    forever
        input menu_choice, "Choose a number and press [ENTER]."
        if menu_choice < 1 OR menu_choice > 3
            errormessage "Choice [" , menu_choice, "]is outside the valid",\
                " range"
        else
            break
        endif
    endfor
endevent
```

Event...EndEvent

Description

The **Event** command indicates the start of a procedure associated with an operator inquiry, payment, an interface response message or printing addition information on the check header or trailer lines. The **EndEvent** indicates the end of the event procedure.

- If the * specifier is present in an Event line, in the Event ID field, then the Event will be executed if the Event types match, regardless of the Event ID. The * specifier affects the following events: **Inq**, **Tmed**, **RxMsg**, and **Final_Tender**. For example, the following **Event** will catch all **Inquire Events**:

```
event inq : *

endevent
```

- It is possible to write an **Event Inq** or **Event Tmed** as an expression. Example:

```
event inq : 5
```

  can be defined as:

```
event inq : ( 2 + 3 )
```

  In addition, variables may also be used, but must be defined as global variables. Example:

```
var guest_inq_number : N5 = 5

event inq : guest_inq_number
```

- The **EndEvent** command cannot be used within a subroutine.

Syntax 1

Event Inq : *number*

Argument	Description
number	corresponds to a pre-defined SIM Inquiry key programmed in the Simphony database, or * to execute the **Event** whenever it is encountered

Remarks 1

- The **Event Inq** command is executed when a SIM Inquiry key is used at a workstation.
- The valid entry for *number* is 1 through 20.

Example 1

This is an example of a standard Inquiry event:

```
event inq : 1
    var menu_choice: n3
    window 3, 23
                .
                .
                .
endevent
```

Syntax 2

Event Tmed : *number*

Argument	Description
number	corresponds to a pre-defined SIM Inquiry key programmed in the Simphony database, or * to execute the **Event** whenever it is encountered

Remarks 2

- The **Event Tmed** command is executed when an ISL Tender key is used at the workstation.
- The Tender *number* must be an object number in the *Tender Media* module. It is required that the Tender Media PMS Option, **Use ISL TMED Procedure Instead of PMS Interface** is enabled, and the workstation must be within a transaction for this Event to work. For a complete explanation, please see “Create a SIM Tender Key” on page 2-14.

Example 2

This is an example of a standard Tender/Media event:

```
event tmed : 10
    var rowcnt : n3
    var deliv_desc[6] : a15
    deliv_desc[1] = "Name:"
    window 6, 43
                .
                .
                .
endevent
```

Syntax 3

Event RxMsg : *event_ID*

Argument	Description
event_ID	the first field in the response message that identifies the event that is expecting that response

Remarks 3

- The first field in a response message is always the *event_ID* and should not be used in any successive **RxMsg** variable.The *event_ID* must begin with a letter A - Z, a - z, or the underline character (_), and it can be up to 255 characters in length.

- When a message has been received from the PMS, the ISL will search the script for an **RxMsg** event whose event type matches the first field in the application_data segment of the message. If ISL encounters a message of the form: Event **RxMsg : *** it will automatically run that event without regard to the PMS message's first field value. This feature is useful for debugging ISL scripts when the message from the PMS may not be correct.

- The **Event RxMsg** command is executed when SIM has been instructed to wait for a response and a response is received from the interfaced system. This event requires that both the **TxMsg** and **WaitForRxMsg** commands be used in another event, in the script file, for the **RxMsg** command to work.

- If the **UseISLTimeOuts** command is used and the PMS does not respond to an ISL message within the timeout period, the ISL will search the script for an **RxMsg** event with an *event_ID* of **_Timeout** (**Event RxMsg** : **_Timeout**). If **_Timeout** is found, ISL will bypass the standard Simphony error messaging and process a user-defined ISL instruction in its place.

- The interface application data message fields are always separated by an ASCII field separator character (1CH).

Example 3a

This is an example of a standard response message event:

```
event inq : 1
    var room_num : a4
    input room_num, "Enter Room Number"
    txmsg "charge_inq",@CKEMP,@CKNUM,@TNDTTL,room_num
                                                        //The first field (charge_inq)
is an
                                                        // example of an identifying
string
                                                        // that the POS might use to
process
                                                        // messsage from the POS.
    waitforrxmsg
endevent
event rxmsg : charge_declined                           //This is one of the PMS
response
                                                        // possibilities
    var room_num : a4
    rxmsg room_num
    exitwitherror "Charge for room ", room_num," declined"
endevent
```

Example 3b

This is an example of an event that is run when the response message is not received within the ISL timeout period:

```
useisltimeouts

event tmed : 10

    var room : N5
    input room, "Enter room number"
    txmsg "CHARGE", room, @tndttl
    waitforrxmsg
endevent
event rxmsg : charge_response
    waitforclear "Posting successful"
endevent
event rxmsg : _timeout
    window 4, 30
    display 2, @center, "PMS is down."
    display 3, @center, "Post to alternate tender?"
    waitforconfirm
    usebackuptender
endevent
```

Syntax 4

Event Final_Tender

Remarks 4

- This event is called after the last tender has occurred, but just before the check is closed. This event is a separate event from the **Event Tmed** event. An **Event Tmed** event is used to post the tender, while the **Event Final_Tender** is used when the check has been completely tendered.

  For example, one could use the **Event Final_Tender** to implement the following features with the ISL:

  - creating a specialized printout of a guest check for which neither the **Event Inq** or **Tmed** can be called when all the check detail is in the check

  - sending log information to a PMS containing all of the check information

- If the **Event Final_Tender** is not present in the script, no error will occur.

- Unlike the **Event Tmed**, when using the **Event Final_Tender** command, a tender does not need to be linked to a PMS by Tender Media PMS Option, **Use ISL TMED Procedure Instead of PMS Interface**.

- When the **Event Final_Tender** is executed, the ISL will execute the event for each script linked to a PMS Computer. For example, if pms1.isl and pms2.isl both include an **Event Final_Tender**, the ISL will process both scripts.

- There is no *event_ID* field for the **Event Final_Tender**.

The **Print_Header** and **Print_Trailer** events, along with the some new SIM system variables (see 6-110 and 6-205) and a specific set of control characters are used to print information on checks and receipts. This information can be printed in the header and/or trailer of Customer Receipts, Guest Checks, and Credit Card Vouchers. This information can include text, bar codes, estimated tip amounts, or any function a SIM script is capable of performing.

Syntax 5

Event Print_Header : *<alpha/numeric >*
Event Print_Trailer : *<alpha/numeric >*

Argument	Description
alphanumeric	corresponds to an entry in the *RVC Descriptors* module in the Enterprise Management Console (EMC)

Remarks

- Control Characters:

```
@@<event ID argument>
```

  Control characters and SIM event(s) are programmed in the *RVC Descriptors* module. The combination of the control characters and the SIM event will call a SIM script, and the additional text or bar code is printed on either the header or trailer.

Example 1

For example, using the event called “est_tip_amt” the Credit Card Voucher Header lines in the *RVC Descriptors* module will be programmed something like this:

```
1 Tip Amount          _____________
2 Estimated Tip Amount:
3 @@est_tip_amt
4
5 Total               _____________
6
7
8 ------------------------------------------------
9          Signature
```

When POS Operations starts printing the credit card voucher trailer, it will print line 1 and 2 as the part of the header, when the event argument (@@est_tip_amt) at line 3 is recognized, POS Operations will call the SIM script.

Example 2

For example, the following portion of a SIM script will be called by the credit card voucher trailer, and will printout the estimated tip amount on the credit card voucher:

```
event print_trailer : est_tip_amt

        format @trailer[1] as "EST TIP AMT     $", ( @ttldue
* 15 ) / 100
         format @trailer[2] as "                        "

endevent
```

After the SIM script is finished, POS Operations will continue printing the remaining lines on the credit card voucher.

Once the SIM script is called, the script will instruct POS Operation what to print and how to format it. The system variables, @HEADER (see page 6-110) and @TRAILER (see page 6-205), support this function.

- The maximum number of SIM events available is the same as the number descriptor lines available in the header and trailer fields. If there are 6 header lines available, then 6 SIM events can be used.

  For example, if printing a CA voucher header (a total of 6 lines) which contained text on line number 1 and 2, then called a SIM event on line 3, that would leave 3 lines available to print information from within the SIM script. For example, one cannot print 5 of 6 lines of a header, then on the 6th line call a SIM script which prints 5 more lines of text. If the SIM script calls for 5 lines, only 1 line will print, as 5 of the 6 lines have already printed.

- The event argument should consist of only letters, numbers, and an underscore (no spaces or punctuation). Also, the first character must be a letter. For example: @@voucher is a valid entry, @@5voucher would be an invalid entry. Maximum length of the descriptor is 30 characters, plus 2 control characters, which is a total of 32.

- More than one event argument (@@) can be embedded in a trailer.

- The @HEADER[] and @TRAILER[] arrays are unique to each event. This means that each event can begin writing to the array starting at index 1, rather than at the next available index. In the example above, both events started formatting at index 1.

- All transaction system variables are still valid in these events. User input is still allowed, as are file operations and display manipulation.

- The events are called when POS Operations is formatting the print data, and not printing it. Therefore, startprint and other SIM commands can be used to generate printouts while the formatting process takes place.

- If the event is not found in the SIM script, then no error is given. The @@ line is ignored.

See Also

Format, **RxMsg**, **TxMsg**, **UseBackupTender**, **UseISLTimeOuts**, **UseSTDTimeOuts**, **Var**, and **WaitForRxMsg** commands

ExitCancel

Description

This command should be used to exit the current script and cancel the current POS tendering operation.

Syntax

ExitCancel

Remarks

This command might be useful if charge posting was denied.

Example

The following example will either allow a check to be tendered to a room charge or prevent the room charge from being posted:

```
event tmed : 9
   var room : a6
   var guest_name : a20

   window 4, 22, "Room Charge"
   displayinput 2, 2, room, "Enter room number"
   displayinput 3, 2, guest_name, "Enter guest name"
   windowinput
   txmsg "room_charge", room, guest_name
   waitforrxmsg
endevent

event rxmsg : post_response
   var status : a10
   var room : a6
   var guest_name : a40

   rxmsg status, room, guest_name
   if status = "accept"
      exitcontinue
   elseif status = "deny"
      errormessage "Room charge denied"
      exitcancel
   else
      call get_more_info( room, guest_name )
   endif
endevent
```

ExitContinue

Description

This command should be used to end the current script and continue processing the POS tendering operation.

Syntax

ExitContinue

Remarks

- This might be useful if tendering should continue after a guest charge is approved.
- Do not confuse the **ExitContinue** command with the **EndEvent** command. **EndEvent** acts as both an Event procedure delimiter and an implicit **ExitContinue**.
- Do not use the **EndEvent** command instead of the **ExitContinue** command.

Example

The following example will either allow a check to be tendered to a room charge or prevent the room charge from being posted:

```
event tmed : 9
   var room : a6
   var guest_name : a20

   window 4, 22, "Room Charge"
   displayinput 2, 2, room, "Enter room number"
   displayinput 3, 2, guest_name, "Enter guest name"
   windowinput
   txmsg "room_charge", room, guest_name
   waitforrxmsg
endevent

event rxmsg : post_response
   var status : a10
   var room : a6
   var guest_name : a40

   rxmsg status, room, guest_name
   if status = "accept"
      exitcontinue
   elseif status = "deny"
      errormessage "Room charge denied"
      exitcancel
   else
      call get_more_info( room, guest_name )
   endif
endevent
```

ExitOnCancel

Description

This command will exit the script when the [Cancel] key or the [Clear] key is pressed after an **Input** command has been issued.

Syntax

ExitOnCancel

See Also

ContinueOnCancel, **ExitCancel**, and **ExitContinue** commands

ExitWithError

Description

This command is used to display an error message and cancel the current POS tendering operation.

Syntax

ExitWithError *error_message*[{*output_specifier*}] [, *error_message*\ [{*output_specifier*}]...]

Argument	Description
error_message	an *expression* displayed in the error banner, usually to instruct the user of a problem; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

The *error_message* is required.

Example

The following script illustrates how this command will display an error if a charge is denied:

```
event rxmsg : charge_declined
   var room_num : a4

   rxmsg room_num

   exitwitherror "Charge for room ", room_num," declined"
endevent
```

FClose

Description

This command closes a file that was previously opened by the **FOpen** command.

Syntax

FClose *file_number*

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened

Remarks

- The *file_number* specified must be a valid file number. That is, it must correspond to a file already opened. Otherwise, an error message will be generated.
- All files are automatically closed at the end of a script.

Example

The following example would open a file, read from it, and then close it:

```
event inq: 1

   var fn : n5
   fopen fn, "/micros/simphony/data/emplist.dat", read
       .
       .
       .
   fclose fn

endevent
```

See Also

FOpen command

FGetFile

Description

This command gets a file from the SIM file service:

Syntax

FGetFile *RemoteFileName, LocalFileName, Status*

Argument	Description
RemoteFileName	relative to "SimDataFiles" directory in \MICROS\Simphony\EGatewayService
LocalFileName	relative to the location of SarOps.exe (...\PosClient\bin) directory on the workstation
Status	will contain result of operation after completion a status of "0" (zero) indicates the file retrieval was successful a status of any non-zero value indicates the file retrieval failed

Remarks

- All files are automatically closed at the end of a script.

See Also

FPutFile command

FLock

Description

This command locks a file to prevent other processes from writing to the file, usually while it is open.

Syntax

FLock *file_number*, [**Preventwrite**] [**And**] [**Preventread**] [**and**]\
[**Nonblock**]

Argument	Description
file_number	identifies the file to be locked; an integer variable which was assigned in the **FOpen** statement when the file was opened
Preventwrite	a mode separator that prevents others from writing to the specified file, while the lock is in place
And	required by syntax if more then one mode separator is issued
Preventread	required by syntax if more then one mode separator is issued
Nonblock	the **FLock** command will return immediately, whether the lock was successful or not

Remarks

- The purpose of this command is to implement cooperative file locking among processes. Since ISL scripts execute in a multiprocessing environment, it may be necessary for one script to write to a file at the same time another needs to read from it. Without any type of synchronization, corrupted data may be read from or written to the file. (Within this explanation, the terms script and process both refer to the POS Operation process which executes the script.)

- The ISL file locking model is based on the file locking model of the underlying Windows operating system. Files can be locked so that other processes cannot read or write that file until a previous lock has been removed.

- As with Windows, file locking can only be used if ***all*** processes accessing the file implement file locking. If one script locks a file, but another chooses to ignore this lock, then the benefits of the lock are lost.

- If the **Preventread** mode is specified with the **FLock** command, all processes which try to lock the file for reading must also wait until the lock is released.
- If the **Preventwrite** mode is specified with the **FLock** command, all other processes which try to lock the file for writing must wait until the current process has released the lock. However, other processes can read the file.
- If the **Nonblock** mode is specified, the script ***must*** check system variable @FILE_ERRNO to determine if the lock was successful or unsuccessful. Please see "FILE_ERRNO" on page 6-98 for the File Access Error Codes.
- It is not possible to lock portions of a file. The entire file must be locked.
- All locks on files are released automatically when the file is closed.
- If the call to **FLock** is executed and another process is busy writing to the file, the command will wait until the lock is released by the other process. For example, assume that there is a file which all ISL scripts need to read. There is also a procedure inside the ISL script, which every so often, needs to update the file (to add new records, for example).
- Locks should be placed on files for only short periods of time. Keeping a file locked for a long time prevents other processes from accessing the file.

Example 1

The following script shows how to lock a file for reading only:

```
event inq : 1

    var fn : N5
    fopen fn, "/micros/simphony/etc/custlist.dat", read
    flock fn, preventwrite
    call read_from_file( fn )
    funlock fn
    fclose fn

endevent
```

Example 2

The following script shows how to lock a file for reading and writing:

```
event inq : 2

   var fn : N5
   fopen fn, "/micros/simphony/etc/custlist.dat", append
   flock fn, preventwrite and preventread
   call write_to_file( fn )
   funlock fn
   fclose fn

endevent
```

The call to **FLock** will wait until all files are done reading.

Example 3

The following script gives an example of the incorrect way of using the **FLock** command; the file is locked while the script waits for input from the user:

```
event inq : 3

   var fn : N5, data : A20
   fopen fn, "/micros/simphony/etc/custlist.dat", append
   flock fn, preventwrite and preventread

   input data, "Enter customer id"

   call write_to_file( fn, data )
   funlock fn
   fclose fn

endevent
```

Example 4

The proper way to implement the script in example 3 would be:

```
event inq : 1

   var fn : N5, data : A20
   input data, "Enter customer ID#"

   fopen fn, "/micros/simphony/etc/custlist.dat", append
   flock fn, preventwrite and preventread
   call write_to_file( fn, data )
   funlock fn
   fclose fn

endevent
```

See Also

FClose, **FOpen**, and **FUnLock** commands

FOpen

Description

This command opens a file for reading or writing.

Syntax

FOpen *file_number*, *file_name*, [**Append**] [**And**] [**Read**] [**And**] [**Write**], [**Local**],[**Unicode**]

Argument	Description
file_number	an integer variable which will be assigned a file number to identify the file
file_name	a string which identifies the file to be opened
Append	a mode separator that appends to an open file
And	required by syntax if more then one mode separator is issued
Read	a mode separator that reads from an open file
Write	a mode separator that writes to an open file
Local	a mode separator that indicates the file is located on the local client workstation (available in SAR only)
Unicode	a mode separator that identifies the file as Unicode (available in SAR only)

Remarks

- The variable *file_number* will be assigned a value of 0 if the operation was unsuccessful. This could occur if the file was opened for reading and did not exist, or the permissions of the file were not set correctly.
- The variable *file_name* must use Windows naming conventions and pathnames. If a file is written to, and does not exist, the file will be created.
- The system variable @FILE_ERRNUM will contain the operating system error code corresponding to the error which occurred when **FOpen** was executed.
- The **Unicode** keyword can be used with, without, before, or after the **Local** keyword. If used without, *do not* include an extra comma separator where the Local keyword would have been.

Example 1

The following statements would open a file and read it:

```
var fn : N5
                              //open a file for reading
fopen fn, "/micros/simphony/data/emplist.dat", read
```

Example 2

The following statement would open a file and append to it:

```
var fn : N5
                              //open a file for appending
fopen fn, "/micros/simphony/log/transact.log", append
```

Example 3

The following statement would open a file and write to it:

```
var fn : N5
                              //create a file for writing
fopen fn, "/micros/simphony/log/ws.log", write
```

Example 4

The following statement would open a file then read and write to it:

```
var fn : N5
                              //open a file for reading and writing
fopen fn, "/micros/simphony/data/emplist.dat", read and write

endevent
```

Example 5

The following script will open a file. If the open was unsuccessful, an error message will display the cause of the error.

```
event inq : 1
   var fn : N5
   fopen fn, "myfile.dat", read
   if fn = 0
      errormessage @FILE_ERRSTR
      exitcontinue
   endif

endevent
```

See Also

FClose command

For...EndFor

Description

These commands are used to implement an iterative loop. The **EndFor** command should always be used to terminate the loop.

Syntax

For *counter* = *start_expression* **To** *end_expression* [**Step** *increment*]

.
.
.
EndFor

Argument	Description
counter	a variable that is incremented by the **For** command
start_expression	the first variable in the *counter*; separated from counter by = sign; it can be one of the following: *user_variable* *system_variable* (N or $ format) *constant* *function*
...=...**To** ...[**Step**...]	required by syntax
end_expression	the last variable in the *counter*; separated from the start expression by the reserved word **To**; it can be one of the following: *user_variable* *system_variable* (N or $format) *constant* *function*
increment	used with the reserved word **Step** to increase or decrease the value of the *counter*; use a negative value to decrease the counter

Remarks

- Normally, the variable in the **For** loop will be incremented by one. If required, the **Step** feature may be used to override this so that the variable may be incremented or decremented by any integer value.

- **For** loops work similar to C and Basic; the **For** loop counter will always increment to the *end_expression* + **1**. A **For** loop will execute when the following conditions are met:

  If **Step** > 0 and counter <= *end_expression*

  If **Step** < 0 and counter >= *end_expression*

  Sample **For** commands:

```
for i = 1 to 10                  // execute 10 times
for i = 1 to 10 step 5           // execute 2 times (i=1,6)
for i = 1 to 10 step -5    // will not execute
for i = 10 to 1            // will not execute
for i = 10 to 1 step -1    // execute 10 times
for i = 10 to 1 step -5    // execute 2 times (i=10,5)
```

Example

The following script will display the current occupant(s) of a room:

```
event rxmsg : display_occupants
   var row_cnt : n3, room_num : a4, number_occupants : n3,
occupant_list[8] : a30
   rxmsg room_num, number_occupants, occupant_list
   if number_occupants > 14
      number_occupants = 14
   endif
   window number_occupants, 38
   for row_cnt = 1 to number_occupants
      display row_cnt, 2, occupant_list[row_cnt]
   endfor
   waitforclear
endevent

//For example, this subroutine will reverse a string
// using the Step feature
sub reverse_string
   var cnt : n3, char : a1, reversed_string : a78
   window 2, len(string_2_reverse) + 2
   display 1, 2, string_2_reverse
   for cnt = len(string_2_reverse) to 1 step -1
      char = mid(string_2_reverse, cnt, 1)
      format reversed_string as reversed_string, char
      display 2, 2, reversed_string
   endfor
endsub
```

See Also

Break and **Forever** commands

ForEver...EndFor

The **ForEver** command provides continuous looping capabilities in a script. The **ForEver** command is generally used when the conditions for terminating the loop are too complex for a **For** command, or may not be known ahead of time. This loop may be broken by executing a **Break** command or by exiting the script (e.g., **ExitCancel** or **ExitContinue**).

Syntax

ForEver

.

.

.

EndFor

Example

The following script will wait for a magnetic card swipe:

```
event inq : 1
   var mag_card_track2_data : a79

   window 1, 28                    // build the window

   forever                         // loop until the user swipes a
                                   // card or presses clear
      displaymsinput 1, 0, mag_card_track2_data{m2, 1, 4, *}, " "
      display 1, 2, "Please swipe your ID card."
      windowinput
      if @MAGSTATUS = "Y"          // we got a swipe
         windowclose               // close the window
         break                     // and exit the loop
      endif
      errormessage "Swipe card or press clear twice"
   endfor
endevent
```

See Also

Break, **ExitCancel**, **ExitContinue**, and **Return** commands

Format

Description

This command is used to concatenate expressions into a string variable.

Syntax

Format *string_variable* [, *field_sep_char*] **As**
expression[{*output_specifier*}], *expression*[{*output_specifier*}]
[, *expression*[{*output_specifier*}],…]

Argument	Description
string_variable	a place holder for text characters such as a *user_variable (string)*
field_sep_char	the character used to separate fields; use the **Chr** function to define the character required
As	required by syntax
expression	an *expression* to be concatenated; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

- If the field separator character is specified, then the first character in the string is used to separate variables within the string.

```
format string as 1, 2, 3          // will create '123'
format string, "," as 1, 2, 3     // will create '1,2,3'
```

- The **Format** command is also used to print information to guest checks, receipts, and credit card vouchers. Two events: Print_Header and Print_Trailer (see page 7-62) are used to support this function.

Example

The following script will construct a *string_variable* containing the current date in the form dd-mm-yy:

```
event inq : 1
   var date : a9

   call get_date_string
endevent
sub get_date_string
   var month_arr[12] : a3
                                          //Listing of all the months
   month_arr[1] = "JAN"
   month_arr[2] = "FEB"
   month_arr[3] = "MAR"
   month_arr[4] = "APR"
   month_arr[5] = "MAY"
   month_arr[6] = "JUN"
   month_arr[7] = "JUL"
   month_arr[8] = "AUG"
   month_arr[9] = "SEP"
   month_arr[10] = "OCT"
   month_arr[11] = "NOV"
   month_arr[12] = "DEC"
   format date as @DAY, "-", month_arr[@MONTH], "-", @YEAR
                          // i.e., 10-NOV-93
endsub
```

See Also

FormatQ and **Split** commands

FormatBuffer

Description

This command will format a string containing non-printable characters into a string that is printable.

Syntax

FormatBuffer *source_string*, *destination_string*

Argument	Description
source_string	a *string_variable* containing non-printable characters
destination_string	a place holder for the *string_variable* containing the printable characters

Remarks

- All printable characters will display as-is. All non-printable characters will be formatted as a two-digit hexadecimal number surrounded by angle brackets.
- This function is generally used to look at data from the PMS.

Example

The following script will convert a string containing a non-printable character into a string that can be displayed:

```
event inq : 1
   var source_s : A30, dest_s : A30

   format source_s as "before ", chr( 27 ), " after"
   formatbuffer source_s, dest_s
   waitforclear dest_s                  //displays 'before <1B>
after'
endevent
```

See Also

Format and **FormatQ** commands

FormatQ

Description

This command is used to concatenate variables into a string. String variables are automatically surrounded by quotes. This feature can be used to create comma-separated lines in ASCII files.

Syntax

FormatQ *string_variable* [, *field_sep_char*] **As**
expression[{*output_specifier*}], *expression*[{*output_specifier*}]...]
[, *expression*[{*output_specifier*}],…]

Argument	Description
string_variable	a place holder for text characters such as a *user_variable (string)*
field_sep_char	the character used to separate fields; use the **Chr** function to define the character required
As	required by syntax
expression	an *expression* to be concatenated; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

- If the field separator character is specified, then the first character in the string is used to separate variables within the string.
- The **FormatQ** command operates in the same way as the **Format** command, except that all strings are automatically quoted. This command is generally used to format lines to a file.

See Also

Format and **Split** commands

FormatRaw

Description

This command allows a SIM script to send up to 2 Kilobytes of raw (un-altered) data to only IDN, Serial, IP, and Bluetooth printers.

Examples of raw data include:

- Barcodes
- QR codes
- Simple text (alphanumeric characters)
- URLS or e-mail addresses

The raw data can be added to appear on guest checks or customer receipts Headers or Trailers.

Syntax

FormatRaw *argument, max_size, data*

Argument	Description
argument	*The argument which replaces the corresponding data in a print command, printer header, or print trailer*
max_size	*The maximum length of the data to be stored in the argument (up to 2 Kilobytes)*
data	*The un-altered raw data that is sent directly to the printer*

Example

```
event INQ : 1

                var qrCodeData: A255
                var qrCodeDataLen : N3
                var rawDataLen: N3


                startprint @CHK
```

```
                format qrCodeData as "This is QR Code data!
www.helloworld.com/index.php, complete our survey and you will get
the next meal free."
                qrCodeDataLen = len(qrCodeData) + 3
                rawDataLen = len(qrCodeData) + 48

                FORMATRAW "ABC", 200, chr(127), rawDataLen, ";", \

chr(13), chr(10), \

chr(27), chr(97), chr(1), \

chr(29), chr(40), chr(107), chr(4), chr(0), chr(49), chr(65),
chr(50), chr(0), \

chr(29), chr(40), chr(107), chr(3), chr(0), chr(49), chr(67),
chr(5), \

chr(29), chr(40), chr(107), chr(3), chr(0), chr(49), chr(69),
chr(48), \

chr(29), chr(40), chr(107), chr(qrCodeDataLen), chr(0), chr(49),
chr(80), chr(48), qrCodeData, \

chr(29), chr(40), chr(107), chr(3), chr(0), chr(49), chr(81),
chr(48), \

chr(27), chr(64)
                printline "@@ABC@@"

               endprint
endevent
```

See Also

PrintLine command and the **@Header** and **@Trailer** system variables

FPutFile

Description

This command puts a file from the SIM file service:

Syntax

FPutFile *RemoteFileName, LocalFileName, Status*

Argument	Description
RemoteFileName	relative to "SimDataFiles" directory in \MICROS\Simphony\EGatewayService
LocalFileName	relative to the location of SarOps.exe (...\PosClient\bin) directory on the workstation
Status	will contain result of operation after completion a status of "0" (zero) indicates the file retrieval was successful a status of any non-zero value indicates the file retrieval failed

Remarks

- All files are automatically closed at the end of a script.

See Also

FGetFile command

FRead

Description

This command reads formatted data from a file.

Syntax

FRead *file_number*, *user_variable* or *list_spec*[, *user_variable* \
or *list_spec*...]

Argument	Description
file_number	identifies the file to be read; an integer variable which was assigned in the **FOpen** statement when the file was opened
user_variable	a *user_variable* which will be assigned the data from the file
list_spec is defined as: *number_records*	 a *user_variable* (integer) containing the number of records to be built from this *string*
field_array[:field_array]	an *array_variable* that will hold one field per record; a field can be split into more than one *array* by separating the *array_variables* using a colon (:)

Remarks

- The file must have been opened in **Read** mode in order to execute this command.
- If the System Variable @STRICT_ARGS is set to 1, then ISL will ensure that the variable count in the **FRead** command line matches the number of fields in the file record. If an incorrect number of fields is specified in the statement or the file is corrupted, then an error message will be generated.

- It is possible to skip over fields in a line by not specifying the variables. For example the third field in the line below would be ignored:

```
fread file_number, variable1, variable2,  , variable4
```

  If a script needs to read only the first few fields in a file, but wishes to ignore the rest of the fields, then it should specify a * in the statement to indicate that no more variables should be assigned to that line. For example, if each line in a file has 20 fields, but only the first three need to be read, the following line will only read the first three. All fields are assigned in the order they occur. The * must be the last element on the **FRead** line.

```
fread file_number, variable1, variable2, variable3, *
```

  This command will assign data to the variables a line at a time. If the line in the file has 10 variables and only 7 variables are specified, then the last 3 are thrown away. They are not read on the next **FRead**.

Example

If a file contains this line:

```
145,"Tooher","Dan"
```

An ISL script uses the following lines to read the file:

```
event inq : 1

   var num:N5, last_name:A20, first_name:A20
      fread file_number, num, last_name, first_name

      //num will be 145
      //last_name will be "Tooher"
      //first_name will be "Dan"

endevent
```

See Also

FClose, **FOpen**, **FReadBfr**, and **FReadLn** commands

FReadBfr

Description

This command reads a block of data from a file.

Syntax

FReadBfr *file_number*, *data*, *count_to_read*, *count_read*

Argument	Description
file_number	identifies the file to be read; an integer variable which was assigned in the **FOpen** statement when the file was opened
data	*string_variable* the *data* block will read into
count_to_read	how much data to read
count_read	how much data was actually read

Remarks

- The file must have been opened in read mode in order to execute this command.
- This command will read data across lines. This command is equivalent to a raw read from a file.

Example

The following script will attempt to read 100 characters:

```
event inq : 1

    var fn : n5
    var data:A100, linesread:N5

    fopen fn, "/micros/simphony/etc/script.isl", linesread
    freadbfr fn, data, 100, linesread
       if linesread <> 100
          errormessage "Tried to read 100 and read ", linesread
          exitcancel
       endif
endevent
```

See Also

FClose, **FOpen**, **FRead**, and **FReadLn** commands

FReadLn

Description

This command reads a line of data from a file.

Syntax

FReadLn *file_number*, *line*

Argument	Description
file_number	identifies the file to be read; an integer variable which was assigned in the **FOpen** statement when the file was opened
line	string variable where the line will read into

Remarks

- The file must have been opened in **Read** mode in order to execute this command.
- This command may be useful if the file being **Read** does not store its data in comma-separated format. For example, the Windows system variable, `%PATH%`, stores its data separated by the (;) character. A script could read the variable and use the **Split** command to access the individual path components.

Example

The following statements would search for a certain line in the /etc/passwd directory:

```
fopen fn, "/etc/passwd", read
while not feof( fn )
   freadln fn, line
   split line, ":", name,, user_id, group_id, *
endwhile
fclose fn
```

See Also

FClose, **FOpen**, **FRead**, and **FReadBfr** commands

FSeek

Description

This command goes to a specified position in the file.

Syntax

FSeek *file_number*, *seek_position*

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened
seek_position	where to position the file pointer (specify the offset of the byte the programmer wants to position to)

Remarks

- Whenever a file is opened, the file pointer is positioned at the start of the file. When data (a line or number of characters) is read or written, the file pointer is positioned at the end of the data that was read or written. The **FSeek** command allows the user to position the file pointer to an arbitrary point in the file so that the next read or write statement will act on the data or position following the new location of the file pointer.

- If a seek position of -1 is specified, the file pointer will be positioned to the end of the file.

Example

The following example gets a number from user, then uses **FReadLn** to read the first field from each line, testing it against the number the user entered. Once the number is found using **FSeek**, the file pointer is positioned at the beginning of the line where the number was found. Then the entire line is read and the first 77 characters displayed for the user.

```
event inq : 1
    var fn : n3
    var fname : a30 = "/micros/simphony/sql.out"
    var line : a200
    var objnum : n6
    fopen fn, fname, read                       //Open the file
    forever
        input objnum, "Enter number to search for"//Get number to search
                                                // for from user
        fseek fn, 1                             // move file pointer
                                                // to beginning of file
        call find_obj( fn, objnum )             //Call the subroutine
        if objnum = 0                           // if 0, no match was
                                                // found, break out
            break

        endif
        freadln fn, line                        //Read the line where
                                                // match found
        window 1, 78                            //Open window
        display 1, 2, mid( line, 1, 77 )        //Display the line
        waitforclear
        windowclose                             //Close the window
        endfor
endevent

sub find_obj( ref fn, ref objnum )
    var current_position : n6
    var found_num : n6
    prompt "Searching, please wait..."
    while not feof( fn )                        //Loop until end of
                                                //file encountered
        current_position = ftell( fn )          //Get the current file
                                                // pointer position
        fread fn, found_num, *                  //Fread the first field
                                                // only from the file
        if found_num = objnum                   //If it matches
                                                //what user entered
            fseek fn, current_position          //Fseek to the beginning
                                                // of the line
            return                              //Exit the subroutine
        endif
    endwhile
    errormessage "Can't find that number"       //If end of file is
                                                // encountered then
    objnum = 0                                  // we didn't find a
                                                // match, tell user, set
endsub                                          // objnum = 0, and return
```

See Also

FClose and **FOpen** commands

FUnLock

Description

This command releases any previous locks on a file.

Syntax

FUnLock *file_number*

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened

Remarks

Closing a file automatically releases any locks on a file.

Example

The following example will lock and unlock one file within an **Event** procedure, while another **Event** procedure attempts to access it.

```
event inq : 1
   var fn : n3
   var fname : a30 = "/micros/simphony/etc/preventread"
   var rite : n5 = 30
   var rote : n5
   var data : a30 = "Some data to write to file"

   fopen fn, fname, read and write
   prompt "Waiting for write access..."
   flock fn, preventread
   fwritebfr fn, data, rite, rote
   waitforclear "File read lock in progress..."
   funlock fn
   waitforclear "File should be unlocked..."
endevent
```

(continued)

```
event inq : 2
   var fn : n3
   var fname : a30 = "/micros/simphony/etc/preventread"
   var reed : n6 = 30
   var red : n6
   var data : a30

   fopen fn, fname, read
   prompt "Waiting for read access..."
   flock fn, preventwrite
   freadbfr fn, data, reed, red
   window 1, 32
   display 1, 2, data
   waitforclear "File write lock in progress..."
endevent
```

See Also

FClose, **FLock**, and **FOpen** commands

FWrite

Description

This command writes formatted data to a file.

Syntax

FWrite *file_number*, *variable1* [, *variable2*][, *variable3*...]

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened
variable n	variables which will be written to the file, where *n* is the number of variables to write

Remarks

- The file must have been opened in **Write** mode in order to execute this command.
- All strings on the **FWrite** line will be enclosed in quotes.
- This function will write one line of data to the file. The line will be terminated with the standard new line character.

Example

The following statements will write a single line of data to 3 different lines:

```
fwrite fn, 1, 500, "line 1"
fwrite fn, 2, 501, "line 2"
fwrite fn, 3, 502, "line 3"
```

The above statements will produce the lines below in the data file:

```
1,500,"line 1"
2,501,"line 2"
3,502,"line 3"
```

See Also

FClose, **FOpen**, **FWriteBfr**, and **FWriteLn** commands

FWriteBfr

Description

This command writes formatted data to a file.

Syntax

FWriteBfr *file_number*, *data*, *count_to_write*, *count_written*

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened
data	string variable to write
count_to_write	how much data to write
count_written	how much data was actually written

Remarks

- The file must have been opened in **Write** mode in order to execute this command.
- This command will write data across lines.

Example

This example reads from a file one character at a time, capitalizes the character as long as it is not in a quoted string, and, if the character was changed, writes the new character back to the position in the file where its lowercase counterpart was found.

```
event inq : 1
    var fn : n3 = 1
    var fname : a40 = "/micros/simphony/etc/temp1.dat"
    var ritecnt : n5 = 1
    var rotecnt : n5
    var char : a20
    var fpos : n6
    var aschar : n3
    var inquotes : n3
    var changed : n3
    fopen fn, fname, read and write          //Open the file for
                                             // read and write
    if fn = 0                                //If fn = 0, file
                                             // couldn't be opened

(continued)
```

```
                    call ferr(fname)
                endif

                while not feof( fn )                    //Loop until the end
                                                        // of file is encountered
                    fpos = ftell( fn )                  //Store the current
                                                        // file pointer position
                    freadbfr fn, char, 1, rotecnt       //Read 1 character
                    aschar = asc( char )                //Get the ascii number
                                                        // for the character
                    if aschar = 34                      // if the character
                                                        // is a quotation mark
                        if inquotes = 1                 // and if it is a
                                                        // closing quote
                            inquotes = 0                //Set inquotes flag to 0
                        else                            // But if it's an opening
quote
                            inquotes = 1                //Set inquotes flag to 1
                        endif
                    endif
                    if not inquotes                     //If this character
                                                        // is not in quotes
                        call capitalize( char, changed ) //Pass it to the subroutine
                    endif
                    if changed = 1                      //If the character was changed
                        fseek fn, fpos                  //Fseek back to position
                                                        // where we found the
character
                        fwritebfr fn, char, ritecnt, rotecnt// and write the character
                                                        // to the file
                        changed = 0                     //Reset the changed flag
                    endif
                endwhile
                fclose fn
            endevent

            sub ferr( ref fname)
            exitwitherror "Can't open file ", fname
            endsub

            sub capitalize( ref achar, ref changed )    //We could also use the
                                                        // UpperCase command
                var aschar : n3 = asc( achar )          // but we'll take the
                                                        // slow difficult route for
fun
                if aschar  > 96 and aschar < 123        //If the character
                                                        // is a lower case alpha
                    aschar = aschar - 32                // get its upper case
equivalent
                    achar = chr( aschar )
                    changed = 1                         // and set the changed flag
                endif
            endsub
```

See Also

FClose, **FOpen**, **FWrite**, and **FWriteLn** commands

FWriteLn

Description

This command writes a line of data to a file.

Syntax

FWriteLn *file_number*, *line*

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened
line	a string variable to write to the file

Remarks

- The file must have been opened in **Write** mode in order to execute this command.
- No quotes will be removed from the string that is written.

See Also

FClose, **FOpen**, **FWrite**, and **FWriteBfr** commands

GetEnterOrClear

Description

This command waits for the operator to press the [Enter] key or the [Clear] key and reports which key was pressed.

Syntax

GetEnterOrClear *input_variable, prompt_expression* \
[{*output_specifier*}][, *prompt_expression*[{*output_specifier*}]...]

Argument	Description
input_variable	an *user_variable* that accepts user input
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

- A value of 0 or 1 is placed in the *input_variable*, depending upon which key is pressed; 0 is placed in the variable if the [Clear] key is used and 1 if the [Enter] key is used.
- The combined length of all *prompt_expressions* must not exceed 38 characters (including spaces); extra characters will be truncated.
- The *prompt_expression* is required.

Example

The following script will wait for either the [Enter] key or the [Clear] key:

```
event inq : 1
    var ent_or_clr : n1
    var ENTER : n1 = 1

    getenterorclear ent_or_clr, "Press ENTER to Inquire, CLEAR to end"
    if ent_or_clr = ENTER
        txmsg "inquiry_1"
        waitforrxmsg
    else
        exitcontinue
    endif
endevent
```

GetTime

Description

This command reads the current time atomically, allowing the script to read *all* of the time and date value, which guarantees that the values will be correct.

Syntax

GetTime [*year*], [*month*], [*day*], \ [*hour*], [*minute*], [*second*], \ [*day_of_week*], [*day_of_year*]

Remarks

- Each variable on the command line corresponds to the time value to read.
- It is not necessary to include each value in the command.

Example 1

```
gettime year, month, day// Get only the date
```

Example 2

```
gettime ,,, hour, minute, second// Get only the time
```

Example 3

```
gettime year, month, day, hour, minute, second// Get
everything but last two
```

If...Else[If]...EndIf

Description

These commands allow conditional execution. The **If** command may be used to compare one expression to another. The **Else** command is used to execute a group of commands when the **If** command's condition is not met. The **ElseIf** command can be used to execute commands when the **If** command's condition is not met and another condition needs to be tested.

Syntax

If *expression* [*operator expression*][**And** | **Or** *expression operator* \
expression...]
.
.
.
Else

or

ElseIf *expression* [*operator expression*][**And** | **Or** *expression* \
operator expression...]
.
.
.
EndIf

Argument	Description
expression	one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
operator	can be one or more Relational Operator
And | **Or**	Relational Operators used to provide additional or alternative conditions

Remarks

- Numeric, currency, alphanumeric, and key variables may be compared. For example, the following usages are valid:

```
if counter < 20
if name = "Richard"
if keyname > @KEY_CLEAR
```

- The *expression* will always be evaluated as true or false; i.e., anything that evaluates to 0 is false and anything that evaluates to non-0 (including a negative) is true. If the operator and second *expression* are left off, the remaining *expression* will still be evaluated in this way.

```
if counter                          //This will be true as long
                                    //as the counter is not 0
```

- Please see "ISL System Variables" on page 6-1.

- It is not considered a fatal error if an **Else** command appears without an **If** command preceding it. An error will occur if a corresponding **EndIf** command is not found.

- The text "Then" is allowed after an **If** or **ElseIf** statement, but it is not required. Although this syntax is legal, it conveys no additional meaning to the If or ElseIf statement in which it used. Example:

```
If i < 4 then                       //Correct
.
.
.
ElseIf i > 10 then                  //Correct
.
.
.
```

Example

The following script will wait for a number entry between 1 and 10:

```
event inq : 9
    var key_pressed : key             //Hold the function key user presses
    var data : a10                    //Hold the number user chooses

    forever
        inputkey key_pressed, data, "Number then Enter, Clear to Exit "

        if key_pressed = @KEY_CLEAR
            exitcontinue
        elseif key_pressed = @KEY_CANCEL
            exitcontinue
        elseif key_pressed = @KEY_ENTER
            if data > 0 and data  <= 10
                waitforclear "You chose" , data, ". Press clear. "
            else
                errormessage "Choose a number between 1" , \
                  " and 10, then press enter"
            endif
        endif
    endfor
endevent
```

See Also

For, **ForEver**, and **While** commands

Input

Description

This command accepts an entry from the operator.

Syntax

Input *input_variable*[{*input/output_specifier*}], *prompt_expression*\
[{*input/output_specifier*}][, *prompt_expression,...*]

Argument	Description
input_variable	a *user_variable* that will store the user's input
{*input/output_specifier*}	one or more of the *input* and *output_specifiers* that determine the format of all input and output fields; see full definition on pages 7-6 through 7-10
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- Prior to changing the *input_variable*, the user's entry will be validated against the field type and optional format definition.
- The combined length of all *prompt_expressions* must not exceed 38 characters (including spaces); extra characters will be truncated.
- The *prompt_expression* is required.
- Magnetic card entry input formats are allowed with the **Input** command.

- If the [Clear] key is pressed during execution of the **Input** command, the script will terminate unsuccessfully. This can have undesired side effects. If a script transacts a successful posting and then uses the **Input** command to get reference information on the posting from the user, the [Clear] key will perform an implicit **ExitCancel**, even though the posting was successful. The following code ensures that the script will not terminate while data is being entered:

```
var user_entry : A20, key_press : key
forever

   inputkey keypress, user_entry, "Enter ref info"
   if keypress = @KEY_ENTER
      break
            //only terminate if ENTER pressed
   endif
endfor
```

Example

The following script accepts a patron number from the user, then transmits it to the PMS for further action:

```
event inq : 1
   var ptrn_no : a8

   input ptrn_no,"Enter Patron Number"  //Get patron number
   txmsg "Inquire_1", ptrn_no           //Send to PMS
   waitforrxmsg                         //Wait for reply
endevent
```

See Also

InputKey command

InputKey

Description

This command accepts an alphanumeric entry from the operator, then stores the entry and the terminating key stroke in separate variables.

Syntax

InputKey *key_variable, input_variable, prompt_expression*\
{*input/output_specifier*}[*, prompt_expression*[{*input*\
/output_specifier}]...]

Argument	Description
key_variable	a *user_variable* key type
input_variable	a *user_variable* that will store the user's input
{*input/output_specifier*}	one or more of the *input* and *output_specifiers* that determine the format of all input and output fields; see full definition on pages 7-6 through 7-10
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- The *key_variable* will be set equal to the terminating key press. In this way, the script can compare the value held by *key_variable* to the @KEY... System Variables, to test for the terminating keystroke the user pressed. See "ISL System Variables" on page 6-1 for more information about the @KEY... System Variables.

- The *prompt_expression* is required.

Example

The following script waits for a number entry between 1 and 9 followed by the [Enter] key. If the [Clear] or [Cancel] key is pressed, it exits the script:

```
event inq : 9
    var key_pressed : key      //Hold the function key user presses
    var data : a10             //Hold the number user chooses

    forever
        inputkey key_pressed, data, "Number then Enter, Clear to Exit "

        if key_pressed = @KEY_CLEAR
            exitcontinue
        elseif key_pressed = @KEY_CANCEL
            exitcontinue
        elseif key_pressed = @KEY_ENTER
            if data > 0 AND data  <= 10
                waitforclear "You chose ", data, ". Press clear. "
            else
                errormessage "Choose a number between 1 ", \
                   "and 10, then press enter"
            endif
        endif
    endfor
endevent
```

See Also

Input command

InfoMessage

Description

This command will display an informational message and continue.

Syntax

InfoMessage *expression*[{*output_specifier*}][, *expression*\
[{*output_specifier*}]...]

Argument	Description
expression	an *expression* to be displayed; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition on pages 7-6 through 7-10

Remarks

The **InfoMessage** command expects one message line to be displayed. However, the workstation displays two lines. The message line to be displayed is broken up between the two logical lines. If the line is too long to be displayed, it will be truncated.

See Also

ErrorMessage

LabelFeedToPeel

Description

This command feeds printed labels to the label peeling position. This only works on the Epson L90 label printer.

Syntax

LabelFeedToPeel

Remarks

- Prior to the **LabelFeedToPeel** command, the printer must be activated within the script using the **StartPrint** command.
- Printing will not begin until the **EndPrint** command is executed.

Example

```
event inq : 1
    var before : A500
    var after : A500
    format before as
chr(&62),chr(&65),chr(&66),chr(&6f),chr(&72),chr(&65),chr(&0A)
    format after as
chr(&61),chr(&66),chr(&74),chr(&65),chr(&72),chr(&0A)
    startprint @ORDR[7]
    printline before
    labelfeedtopeel
    printline after
    endprint
endevent
```

LineFeed

Description

This command will line feed the selected printer. The number of line feeds is optional.

Syntax

LineFeed [*number_of _line_feeds*]

Argument	Description
number_of_line_feeds	an *expression* which defines the number of line feeds

Remarks

- Prior to the **LineFeed** command, the printer must be activated within the script using the **StartPrint** command. To determine the line feeds required for the printer, refer to the table that defines ISL Printers on page 7-163.
- Printing will not begin until the **EndPrint** command is executed.
- A line feed is automatically executed after the **PrintLine** command is issued.

Example

```
event inq : 1
    var kitchen_msg[13] : a20
    var sender_name : a20
    var rowcnt : n3
    window 14, 22                                   //Display the window
    displayinput 1, 2, sender_name, "Enter your name"
                                                    //Accept users name
    for rowcnt = 1 to 13                            //Have user input the message
        displayinput rowcnt + 1, 2, kitchen_msg[rowcnt], "Enter kitchen message"
    endfor
    windowedit 1                                    //Only save or cancel will
                                                    // end input
    startprint @ordr1                               //Start the print job at
                                                    // remote printer1

        printline "============================"
        printline "Message from ", sender_name
        printline "============================"
        for rowcnt = 1 to 13
            if len(kitchen_msg[rowcnt]) > ""
                printline kitchen_msg[rowcnt]
            endif
        endfor
        printline "======= END MESSAGE ========"
        linefeed 5
    endprint
endevent
```

See Also

EndPrint, Printline, and **StartPrint** commands

ListDisplay

Description

This command is used to display a list (array) variable within a window. This command is useful when displaying the contents of an array variable that contains data received from a PMS, such as a list of names.

Syntax

ListDisplay *row, column, list_size, array_variable*

Argument	Description
row	the integer *expression* specifying the screen *row* within the defined window where the first *array_variable* entry will be displayed
column	the integer *expression* specifying the screen *column* within the defined window where the first *array_variable* entry will be displayed
list_size	the number of *array_variable* entries to display
array_variable	the name of the *user_variable* that holds the matrix of values to be displayed

Remarks

- The **Window** command must precede this command.
- It is acceptable to set *list_size* equal to 0, but if this is done, nothing will display. If the *list_size* is less than zero, an error will occur.
- Each entry will be placed on a separate line directly beneath the previous.

Example

The following script will display an employee list:

```
event rxmsg : emp_list
   var emp_list_size : n3
   var emp_list_array[14] : a40

   rxmsg emp_list_size, emp_list_array[   ]
   window 14, 42
   listdisplay 1, 2, emp_list_size, emp_list_array
   waitforclear
endevent
```

See Also

Window command

ListInput

Description

This command is used to display a list (array) variable within a window at the workstation, then waits for the operator to select an item from the list.

Syntax

ListInput *row, column, list_size, array_variable, input_variable,\ prompt_expression*[{*output_specifier*}]

Argument	Description
row	the integer *expression* specifying the screen *row* within the defined window where the first *array_variable* entry will be displayed
column	the integer *expression* specifying the screen *column* within the defined window where the first *array_variable* entry will be displayed
list_size	the number of *array_variable* entries to display
array_variable	the name of the *user_variable* that holds the matrix of values to be displayed
input_variable	an *array_variable* or *user_variable* that accepts user input
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-11

Remarks

- The **Window** command must precede this command.
- Each list entry is displayed with a selection number starting at 1. The selection numbers 1 to 9 are preceded by a space. Selection numbers are followed by a period, then a space, then the list entry. For this reason, the window drawn must be at least four columns wider than the longest item in the list. Each list entry is placed on a separate line. The user's entry is placed in the *input_variable* and is validated against the number of items in the list.
- The *prompt_expression* is required.
- It is acceptable to set *list_size* equal to 0, but if this is done, nothing will display. If the *list_size* is less than zero, an error will occur.

Example

The following script receives and displays a list of guests from the PMS, allows the user to choose one from the list, and then transmits the user's choice to the PMS for further processing:

```
event rxmsg : room_inquire
    var rm_guest[14] : a20                    //Quest names array
    var rm_num : a6                           //Room number
    var list_size : n3                        //Number of array items
    var user_choice : n3                      //Quest number user chooses
    rxmsg rm_num, list_size, rm_guest[ ]    // receive message from POS


    window list_size, 24, "Guests- Room #", rm_num
    listinput 1, 1, list_size, rm_guest, user_choice, "Choose a guest"

    txmsg "guest inquiry", rm_num, user_choice
                                              //Ask for info from PMS
    waitforrxmsg                              // on guest user chooses
endevent
```

See Also

Window command

ListInputEx

Description

This command is used to display a list and get an operator selection. This command is the same as **ListInput**, but it does not provide a WROW or WCOL variable.

Syntax

ListInputEx *row, column, list_size, array_variable, input_variable,\ prompt_expression*[{*output_specifier*}]

Remarks

- The **Window** command must precede this command.
- Each list entry is displayed with a selection number starting at 1. The selection numbers 1 to 9 are preceded by a space. Selection numbers are followed by a period, then a space, then the list entry. For this reason, the window drawn must be at least four columns wider than the longest item in the list. Each list entry is placed on a separate line. The user's entry is placed in the *input_variable* and is validated against the number of items in the list.
- The *prompt_expression* is required.
- It is acceptable to set *list_size* equal to 0, but if this is done, nothing will display. If the *list_size* is less than zero, an error will occur.

Example

The following script receives and displays a list of guests from the PMS, allows the user to choose one from the list, and then transmits the user's choice to the PMS for further processing:

```
event rxmsg : room_inquire
    var rm_guest[14] : a20                  //Quest names array
    var rm_num : a6                         //Room number
    var list_size : n3                      //Number of array items
    var user_choice : n3                    //Quest number user chooses
    rxmsg rm_num, list_size, rm_guest[ ]    // receive message from POS


    window list_size, 24, "Guests- Room #", rm_num
    listinputex 1, 1, list_size, rm_guest, user_choice, "Choose a guest"

    txmsg "guest inquiry", rm_num, user_choice
                                            //Ask for info from PMS
    waitforrxmsg                            // on guest user chooses
endevent
```

See Also

ListInput and **Window** command

ListPrint

Description

This command will print a list on the selected printer.

Syntax

ListPrint *list_size, array*

Argument	Description
list_size	the number of *array_variable* entries to be printed
array_variable	the name of the *user_variable* that holds the matrix of values to be printed

Remarks

- The **StartPrint** and **EndPrint** commands are required when using the **ListPrint** command.
- It is acceptable to set *list_size* equal to 0, but if this is done, nothing will display. If the *list_size* is less than zero, an error will occur.

Example

The following script receives a list of directions from the PMS that describes how to get from the property to another location and prints them at the UWS's check printer:

```
event rxmsg : directions

    var directions[50] : a35                //Our direction array
    var list_size : n3                      //Number of array items
    rxmsg list_size, directions[  ]         //Here's the message from PMS
    startprint @CHK                         //Print at the check printer
        listprint list_size, directions //Print the list
    endprint

endevent
```

See Also

StartPrint and **EndPrint** commands

LoadDbKybdMacro

Description

This command loads a keyboard macro that is pre-defined in the Simphony database. The macro will execute when transaction processing successfully resumes.

There is another keyboard macro command available: **LoadKybdMacro**, which uses a script-defined keyboard macro.

Syntax

LoadDbKybdMacro *numeric_expression*

Argument	Description
numeric_expression	an *expression* that requires a number; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- The macro is referenced by its object number.
- Only integer variables can be used to run a pre-defined macro.
- If more than one **LoadDbKybdMacro** command is used in the same event, only the last command will be used when transaction processing resumes.

Example

The following script will load a macro from the Simphony database that will add two menu items to a guest check currently open at the workstation:

```
event inq : 1

   loaddbkybdmacro 1

endevent
```

See Also

LoadKybdMacro command

LoadKybdMacro

Description

This command passes keystrokes to the workstation; these keystrokes will be executed when the script event terminates. There are a variety of ways to specify script-defined macros, as listed and described below.

Syntax

LoadKybdMacro *key_expression*[, *key_expression*,...]

Argument	Description
key_expression	an *expression* that can be one of the following: *key_type:key_number* **Key** *function*

Remarks

- For pre-defined database macros, see **LoadDbKybdMacro**.
- A script-defined macro is one that the script writer constructs using key function codes. These key function codes can be represented by any combination of the following methods:
- Key pairs—A *key_pair* in Simphony is designated by the *key_type* and a *key_number* separated by a colon. For example, the number 1 on the numeric keypad is represented by the *key_pair* 9 : 1, where 9 (the *key_type*) represents the Keypad and where 1 (the *key_number*) represents Numeric 1. Therefore, one way to load 123 and the [Enter] key would be:

```
loadkybdmacro 9:1, 9:2, 9:3, 9:12
```

- The **Key** function takes as its argument a *key_pair* separated by a comma and returns a key function code. This comma-separated *key_pair* could be represented by two comma-separated variables. **Key** function - The **Key** function takes as its argument a *key_pair* separated by a comma and returns a key function code. This comma-separated *key_pair* could be represented by two comma-separated variables.

  For example:

```
var key_type : Key = 9
loadkybdmacro key(key_type, 1), key(key_type, 2),\
    key(key_type, 3), 9:12
```

- @KEY... System Variables—System Variables that begin with @KEY... are special key type variables. These keys can be used to test for and represent movement keys and keys like [Enter] and [Clear]. The @KEY... System Variables can also be used in the **LoadKybdMacro** command.

  To continue the example:

```
loadkybdmacro 9:1, 9:2, 9:3, @KEY_ENTER
```

- User-defined variables with the type Key—User-defined variables with the type Key can be assigned key function codes using the **Key** function or with @KEY... System Variables. Another way to load the keyboard macro:

```
var key_1 : Key                    //Declare variables as
                                   //Key types
var key_2 : Key
var key_3 : Key
var Enter : Key

key_1 = key(9,1)                   //Assign them with
                                   //key()function
key_2 = key(9,2)
key_3 = key(9,3)
Enter = @KEY_ENTER                 // and system variable

loadkybdmacro key_1, key_2, key_3, Enter
```

- If more than one **LoadKybdMacro** command is used in the same event, they will be run in order of appearance when transaction processing resumes, as if they were one large macro.

- Macro keys will remain defined during script processing and will execute once a script is complete, not when an event is complete.

Example 1

In the following script, there will be three [Enter] keys that will be run once the script completes:

```
event inq:1
    loadkybdmacro @KEY_ENTER              //first enter key
    txmsg "inq_1_request"
    waitforrxmsg
endevent

event rxmsg : inq_1_reply
    loadkybdmacro @KEY_ENTER              //second enter key
    txmsg "inq_2_request"
    waitforrxmsg
endevent

event rxmsg : inq_2_reply
    loadkybdmacro @KEY_ENTER             //third enter key
endevent
```

See Also

- **LoadDbKybdMacro** command; **Key** function
- “Key Types, Codes, and Names”

LowerCase

Description

This command is used to convert a string variable to lower-case.

Syntax

LowerCase *string_variable*

Argument	Description
string_variable	the string that will be changed; can be any *user_variable* (string)

See Also

UpperCase command

MakeAscii

Description

This command will transfer the data from a *source_string* into a *destination_string*, stripping out any non-ASCII or non-printable characters from the *source_string*.

Syntax

MakeAscii *source_string*, *destination_string*

Argument	Description
source_string	a *string_variable* containing both ASCII and non-ASCII characters
destination_string	a place holder for the *string_variable* containing only ASCII characters

Example

The following script will convert a string containing both ASCII and non-printable characters into a string that contains only ASCII characters:

```
event inq : 1
   var string1 : A20
   var string2 : A20

   format string1 as "ABC", chr(1), "DEF"
   makeascii string1, string2    //string2 will be "ABCDEF"
```

MakeUnicode

Description

This command will transfer the data from a *source_string* into a *destination_string*, stripping out any non-printable characters from the *source_string*.

Syntax

MakeUnicode *source_string*, *destination_string*

Argument	Description
source_string	a *string_variable* containing non-printable characters
destination_string	a place holder for the *string_variable* containing only non-printable characters

Example

The following script will convert a string containing non-printable characters into a string that contains only printable characters:

```
event inq : 1
   var string1 : A20
   var string2 : A20

   format string1 as "μαρ"
   makeunicode string1, string2  //string2 will be "MAP"
```

Remarks

This command is only available on SAR Ops.

Mid

Description

This command is used to set all or some part of one string variable equal to another string variable.

Syntax

Mid (*string_variable, start, length*) = *replacement_string*

Argument	Description
string_variable	the string that will be changed; can be any *user_variable* (string)
start	the character position within the s*tring_variable*, where the *replacement_string* will begin to overwrite
length	the number of characters in the *string_variable* that will be changed using the *replacement_string* to the right of the equal sign
=	required by syntax
replacement_string	the string containing the characters used for the replacement can be any *user_variable* (string)

Remarks

- The parentheses are required.
- The first character in the string is always 1. If the *start* or *length* is less the 0, then an error will occur.

- If the *start* is greater than the *string_variable* itself, no data will be assigned. If the *length* is greater than the room left to assign the *string_variable*, then the data will be truncated.

```
var string : A10

string = "this short"
// In this command, the length exceeds room left.
// Only first 5 letters of ("long string") are used,
// overwriting the last five characters of string.
// In this operation, string would become "this long".

mid( string, 6, 10 ) = "long string"


// In this command, this starting position is greater
// than length of string.
// 0 characters are overwritten.
mid( string, 25, 10 ) = "long string"
```

- Do not confuse this command with the **Mid** function.

Example

The following script will replace the first three letters of the variable "string" with the string "NEW":

```
event inq : 1

   var string: a10 = "OLD STRING"

   waitforclear string           //Prompt will show "OLD STRING"
   mid(string, 1, 3) = "NEW"     //Change OLD to NEW
   waitforclear string           //Prompt will show "NEW STRING"
endevent
```

MSleep

Description

This command tells the script to sleep for the requested number of milliseconds.

Syntax

MSleep *milliseconds*

Example

```
event inq:2

    var seconds:N5

    input seconds, "Enter number of seconds to sleep"

    prompt "Sleeping"

    msleep seconds*1000

    waitforenter "Done waiting ", seconds*1000, " seconds."

endevent
```

PopUpIslTs

Description

This command displays a pop-up touchscreen defined by the **SetIslTsKey** command.

Syntax

PopUpIslTs

Remarks

After a touchscreen has been displayed, its keys remain defined until cleared by the **ClearIslTs** command.

Example

The subroutine below first clears any previously defined touchscreen keys and displays two touchscreen keys, [YES] and [NO], which are defined by the **SetIslTsKey** command. This subroutine displays these keys using the **PopUpIslTs** command, as the operator is issued a prompt by the system, and captures the operator's input.

```
sub get_yes_or_no( ref answer, var prompt_s:A38 )
   var keypress : key
   var data : A20

   clearislts
   setisltskey 2, 2, 4, 4, 3, @KEY_ENTER, "YES"
   setisltskey 2, 6, 4, 4, 3, @KEY_CLEAR, "NO"
   popupislts

   inputkey keypress, data, prompt_s
   if keypress = @KEY_ENTER
      answer = 1
   else
      answer = 0
   endif
endsub
```

See Also

ClearIslTs, **DisplayIslTs**, and **SetIslTsKey** commands

PrintLine

Description

This command prints a line on the selected printer defined in the **StartPrint** command. The print information can be provided as text or by referencing a variable field.

This command may also be used to print Binary Data (used to output information such as barcodes, rotated text, emphasized print, etc.) from within a SIM Script.

Syntax

PrintLine *expression*[{*output_specifier*}] *or directive*\
[, *expression*[{*output_specifier*}] *or directive*]...]

Argument	Description
expression	an *expression* that represents the string to be printed: it may one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
directive	specific instructions that affect the color, width, and justification of the printed characters; see full definition on page 5-7
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-11

Remarks

- This command will affect the printer selected using the **StartPrint** command. For a complete list of available printers, see the table provided in the detail description of the **StartPrint** command.
- If printing was started using the **StartPrint** command, then only the first 32 characters in the **PrintLine** command are significant.
- The **PrintLine** command sends a carriage return/line feed to the printer after every line.

- The **PrintLine** command may be used to format characters to a certain position (i.e., left, right, or center justified, etc.). Please see "Using Format Specifiers" on page 7-5.

- Control characters can be printed on the **PrintLine** using the **Chr** function. For example, one can send a command to print a section of a line in emphasized style. The IBM proportional control code ESC 69 turns emphasized print on, and ESC 70 turns emphasized print off:

```
printline "A line with ", chr(27), chr(69), "emphasized"\
  "print", chr(27), chr(70), " on it."
```

  The only control code not printable using the **Chr** function is the ASCII 0 (NUL). To print a NUL, use the following sequence:

```
chr( 16 ), chr( 48 )
```

- For example to output Binary Data, the following script sends 3 lines of text to the printer connected to the PCWS, and will send the proper control codes for turning on emphasized print for the 2nd line. Note that the emphasize on/ off codes are on the same line.

```
startprint @chk
    printline "This is a normal line"
    printline chr(27), "E1", "This is emphasized", chr(27), "E0"
    printline "This is back to normal"
endprint
```

- The workstation application will decode the binary data, and pass it through to the printer.

- 'Binary data' or 'control codes' are all ASCII characters below 32 (20 hex). For example, chr(31) is a control code, but chr(32) is not a control code. All characters from 32 to 255 are considered printable characters.
  One exception to the control code format is that ASCII NULs cannot be printed using chr(0). If a chr(0) needs to be printed, then the system variable @nul should be used instead. For example:

```
printline chr(27), "E", @nul, "Normal print"
```

  Only 32 printable characters can be sent on one line. However, 48 characters can be placed on the line. Each control character requires 2 characters. Therefore, only 24 control codes can be placed on one printline (24 x 2 => 48). However, non-control codes require only one character. Since not all printer commands are all composed of binary data, this limitation should not present a problem.

- When using the SIM Print_Header or Print_Trailer commands (see page 7-62, the control codes will also be formatted properly to the printer when SIM provides the data using the @header[] and @trailer[] system variables. Though each line is supposed to have 32 characters of data, up to 48 characters can be formatted. However, the rule that each control code requires two characters is still in effect.

Example

The following script constructs and prints a date string bordered by hash marks at the workstation's check printer:

```
sub print_date
   var date : a9
   var hash_mark : a24           //Use format to build a date
                                 // string
   format date as @DAY, "-", month_arr[@MONTH], "-", @YEAR

   setstring hash_mark, "="
   startprint @CHK               //Print what follows at this UWS
                                 // check printer
      printline hash_mark        //Print the date string
      printline @DWON, @REDON, date{=16}// double-wide, in red
      printline hash_mark        // and centered between
                                 // hash marks
   endprint
endsub
```

See Also

StartPrint command and the **Chr** function

Prompt

Description

This command is used to display an operator prompt on the prompt line at the workstation.

Syntax

Prompt *expression*[{*output_specifier*}] [, *expression*\ [{*output_specifier*}]...]

Argument	Description
expression	an *expression* that represents the prompt for the user: it may one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-11

Remarks

- The combined length of all *expressions* must not exceed 38 characters (including spaces); extra characters will be truncated.
- The **Prompt** command should only be used when a time-consuming piece of code is to be executed. This command will inform the user that the UWS is busy. All commands that require user input display their own prompts.

```
prompt "Posting unsuccessful"
              // displayed but erased immediately
waitforclear
              // "Press clear to continue" will display
```

Example

The following script receives a list of guests from the PMS, calls a subroutine that sorts the list, then calls another that displays the list:

```
event rxmsg : guest_list
    var num_guests : n3
    var guest_list[100] : a20

    rxmsg num_guests, guest_list[  ]                 //Get the guest array
from PMS
    prompt "Sorting guest list - please wait..."//Tell the user
    call sort_list                                   //Call routine that
sorts the
                                                     // list
    call display_guests                              //Call routine that
displays
                                                     // the list
endevent
```

ProRate

Description

This command is used to indicate to the system that prorated itemizers are to be used. Prorated itemizers are required for some Property Management Systems.

Syntax

ProRate

Remarks

- If the **ProRate** command is encountered either inside or outside an Event procedure, all itemizers will be prorated for the duration of the ISL script. The following system variables are prorated:

  @SI[]
  @DSC
  @SVC
  @AUTOSVC
  @TAX[]

  Please refer to "ISL System Variables" for more information.

- Prorated itemizers are useful if the PMS is posting sales, discounts, tax, etc. during the charge posting operation. When prorated totals are used, the totals reflect the Current Payment's share of a guest check. If the Current Payment is voided, the totals will have the reversed polarity to reflect this. The only exception (i.e., the total that is not prorated) is a charged tip, which will always be completely attributed to its associated payment. This mode is supported by Simphony.

The following equations can be used by the PMS PC to determine the Current Payment Total of the prorated transaction being posted:

Current Payment Total = + (+) Sales 1 Total
+ (+) Sales 2 Total
+ (+) Sales 3 Total
+ (+) Sales 4 Total
+ (+) Sales 5 Total
+ (+) Sales 6 Total
+ (+) Sales 7 Total
+ (+) Sales 8 Total
+ (+) Sales 9 Total
+ (+) Sales 10 Total
+ (+) Sales 11 Total
+ (+) Sales 12 Total
+ (+) Sales 13 Total
+ (+) Sales 14 Total
+ (+) Sales 15 Total
+ (+) Sales 16 Total
+ (-) Discount Total
+ (+) Service Charge Total (kybd)
+ (+) Service Charge Total (auto)
+ (+) Tax 1 Total (if non-VAT)
+ (+) Tax 2 Total (if non-VAT)
+ (+) Tax 3 Total (if non-VAT)
+ (+) Tax 4 Total (if non-VAT)
+ (+) Tax 5 Total (if non-VAT)
+ (+) Tax 6 Total (if non-VAT)
+ (+) Tax 7 Total (if non-VAT)
+ (+) Tax 8 Total (if non-VAT)

The Previous Payment Total is also provided.

In some jurisdictions, the prorated calculations will result in inexact tax totals. This occurs because of rounding errors associated with proration and the methods required to compute tax. As an example, consider three guests paying a $10.00 check which includes $1.00 tax. The first two guests will be charged $0.33 tax and the third $0.34 tax (the rounding adjustment is included in the last total). These situations are unavoidable; if complete accuracy is required, a Split Check operation should be performed and the remaining checks (after the Split Check) should be individually posted to the PMS.

See Also

@PRORATETND system variable

Notes

- In the formulas on the previous page, the first arithmetic operator indicates the operation that the PMS should perform on the total field. The second arithmetic operator (in parentheses) indicates the normal sign of the total field as presented in the message data block by the POS system.

- If Value Added Tax (VAT) is used, the tax totals represent the total sales amounts (inclusive of VAT) for each of the VAT tax types and **must not** be included in the Transaction Total or Current Payment Total equations.

- In non-prorated mode, if the Current Payment Amount field in the message data block is less than the computed Transaction Total (above), then a partial amount has been tendered.

- If U.S. inclusive tax is used, the tax total associated with this rate will be zero.

QueueMsg

Description

This command holds PMS messages in the Simphony database queue until the PMS is online.

Syntax

QueueMsg pms_number, *expression*[{*output_specifier*}]
[, *expression*[{*output_specifier*}] \...]

Argument	Description
expression	an *expression* that will be queued; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

[Retain/Discard]GlobalVar

Description

These commands instruct the ISL to save all global variable values in between transactions, or to discard them.

Syntax

RetainGlobalVar

or

DiscardGlobalVar

Remarks

- These commands are global, which means they remain in affect until the alternative command is used or until the script file has been changed (i.e., until the script file is opened for edit and closed).
- The default action is to discard the global variable values after each event.

Example

The following example could be used to count the number of times Tender #1 has been used. This value will be retained and incremented until the script is changed.

```
retainglobalvar

var numtnd : n5                  //Numtnd is retained until POS
                                 // Operations is Shut Down or
                                 // Reloaded.
event tmed : 1

numtnd = numtnd + 1
          .
          .
          .
endevent
```

Return

Description

This command is used to return from a subroutine procedure prior to reaching the end of the routine (**EndSub**). It is provided as a means of breaking out of the subroutine under certain conditions.

Syntax

Return

Remarks

The **Return** command is not allowed in an Event procedure.

Example

The following example shows a subroutine that takes as arguments the file number of an open file and a search number. It tests the first field in each line for a number that matches the search number. If a match is found, the subroutine **Returns** without executing the rest of the commands in the subroutine. If no match is found before the end of file is encountered, the subroutine exits normally.

```
sub finfnd_obj( ref fn, ref search_num )
    var current_position : n6
    var found_num : n6
    prompt "Searching, please wait..."
    while not feof( fn )                        //Loop until end
                                                // of file encountered
        current_position = ftell( fn )          //Get the current
                                                // file pointer position
        fread fn, found_num, *                  //Fread the first field
                                                // only from the file
        if found_num = search_num               // if it matches
                                                // what user entered
            fseek fn, current_position          //Fseek to the beginning
                                                // of the line
            return                              // exit the subroutine
        endif
    endwhile
    errormessage "Can't find that number"//If end of file
                                                // is encountered then
    search_num = 0                              // we didn't find a
                                                // match, tell user, set
endsub                                          // search_num = 0,
                                                // and return
```

See Also

If, **Sub**, and **While** commands

ReTxMsg

Description

This command retransmits a message. When used in place of the TxMsg command, the retransmit flag is set to “R” and the sequence number is not incremented.

Syntax

ReTxMsg

Remarks

This command should only be used with the **UseISLTimeOuts** command.

See Also

UseISLTimeOuts command

RxMsg

Description

This command defines the format of the data segment of an interface response message by specifying a variable name for each piece of data it receives. The **RxMsg** command will assign the values in the Applications Data Segment of the variables specified in the **RxMsg** statement.

Syntax

RxMsg *user_variable* or *list_spec*[, *user_variable* or *list_spec*...]

or

RxMsg _Timeout

Argument	Description
user_variable	a *user_variable* which will be assigned data
list_spec is defined as: *number_records*	a user integer *variable* containing the number of records to be built from this *string*
field_array[:*field_array*]	an *array_variable* that will hold one field per record; field can be split into more than one *array* by separating the *array_variables* using a colon (:)

Remarks

- The **Event RxMsg** command is executed when a response is received from the interfaced system. The Event must contain an **RxMsg** command. The first field in a response message is always the **Event** ID and should not be declared using this command. The **RxMsg** command defines all the fields (and their order) following the **Event** ID field. The interface message fields are always separated by an ASCII field separator character (ASCII 1CH). The variable fields must have been previously declared using the **Var** command.

- The *user_variable* can be up to 255 characters in length and must begin with a letter A - Z, a - z, or the underscore character (_). It may include any character in the range A - Z, 0 - 9, and the underscore character.

- If the System Variable @STRICT_ARGS is set to 1, then ISL will ensure that the variable count in the **RxMsg** command matches the number of fields in the file record. If an incorrect number of fields is specified in the statement or the file is corrupted, an error message will be generated.

- This Event requires that both the **TxMsg** and **WaitForRxMsg** commands be used in another event in the script for the **RxMsg** command to work.

- If the **UseISLTimeOuts** command is used and the PMS does not respond to an ISL message within the timeout period, the ISL will search the script for an **RxMsg** event with an **Event** ID of **_Timeout**. If **_Timeout** is found, ISL will bypass the standard Simphony error messaging and process a user-defined ISL instruction in its place.

- If the **RxMsg** command specifies more variables than are present in the applications data segment, no error will occur. The extraneous variables on the command will retain their previous value. If the number of fields in the Applications Data Segment exceeds the number of variables in the **RxMsg** command, no error will occur. The command will execute successfully and the extraneous data in the message will be thrown away.

- It is not possible to execute an **RxMsg** if no message has been received. It is also not possible to execute multiple **RxMsg** commands within one **RxMsg** Event. Once the first **RxMsg** command executes, any subsequent **RxMsg** command will cause an error.

- This command is related to the **Split** command. While the **Split** command works with any string buffer and any field separator, the **RxMsg** command assumes the PMS message and the ASCII field separator character.

Example

```
event inq : 1
    var room_num : a4
    input room_num, "Enter Room Number"
    waitforrxmsg
    txmsg @CKEMP,@CKNUM,@TNDTTL,room_num
endevent
event rxmsg : charge_declined
    var room_num : a4
    rxmsg room_num
    exitwitherror "Charge for room ", room_num," declined"
endevent
```

See Also

Event, **Split**, **TxMsg**, **UseISLTimeOuts**, **Var**, and **WaitForRxMsg** commands

SaveChkInfo

Description

This command saves a type of check detail known as check information detail lines in the Guest Check files.

Syntax

SaveChkInfo *expression*[{*output_specifier*}][*, expression*\
[{*output_specifier*}]]...]

Argument	Description
expression	an *expression* that represents the information to be saved; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- Like other types of guest check detail, i.e., totals and definitions, guest check information detail lines are only stored in the Guest Check files (stored in workstations) temporarily and cleared upon closing a guest check. As a consequence, this detail only has short-term value unless it is also written to a third-party database or to Closed Check files.

    - **Closed Check Files**—Once stored in these files, guest check information detail can be exported to an ASCII comma-separated file using the Simphony Data Access Service or check detail can be written to a file using the ISL file I/O operations. Then, an external program can be created to extract this information from these files and manipulate the information for a variety of purposes.

    - **Third-party System**—If this information is captured by a third-party system, like a delivery system, accessing the information depends on the resources of that particular system.

- The **SaveChkInfo** command is issued once for each line of check information detail written to the check. The process goes like this:
    - The **first** occurrence of **SaveChkInfo** in an event writes to the first check information detail line.
    - Each **subsequent call** to **SaveChkInfo** writes to the next check information detail line.
- Check information detail is not actually written to the check detail until the script terminates.
- Check information detail can be overwritten by using **SaveChkInfo** in another event.

How to Capture and Print Check Information Detail

Introduction

Both SIM and Simphony must be used in the following ways to accomplish these tasks:

- A SIM script must be designed to collect and save the check information detail, and
- Specific Simphony programming must be enabled to print and display the check information detail to the specifications of the establishment.

Designing the SIM Script

The SIM must be used to capture and save check information detail. That is, the script must be designed so that an operator can input check information detail and save it, if necessary, from the System Unit.

When designing the script, keep the following rules in mind:

- **Rule:** Do not issue **SaveChkInfo** in a single event more times than the number of allocated check information detail lines; doing so will result in an error.
- **Rule:** Only 24 characters per check information detail line may be written with **SaveChkInfo**.

The following steps are the sequence of events that the script might execute in order to capture and save check information detail:

1. Begin a check.

2. Draw a window.
3. Prompt the operator to enter name and address information.
4. Issue the **SaveChkInfo** command to save the input data.

Programming the Simphony Database

Whether check information detail lines will print and where they will print on guest checks and remote output is determined by specific Simphony database programming, not the script. In order to print this detail and save it in the Guest Check file:

- check information detail lines must be allocated in Simphony
- programmed to print

For example, one can program the Simphony to print the check information detail captured above or after the guest check header, or after the guest check trailer. In the example of the delivery system, the restaurant programmed the check information detail lines to print before the guest check header.

Operational Considerations

Voiding Check Information Detail

Unlike other types of check detail, check information detail cannot be voided from a check using any Simphony void procedures or modified using the Edit Closed Check function.

Example

The ISL event below initiates a procedure for collecting and saving the following customer information for a fictional delivery system at a restaurant: last and first name, telephone number, street address, and up to four lines for directions.

```
event inq : 1
   var field_name[8] : a24
   var customer_info[8] : a24
      field_name[1] = "Last: "
      field_name[2] = "First: "
      field_name[3] = "Phone: "
      field_name[4] = "Addr: "
      field_name[5] = "D1: "
      field_name[6] = "D2: "
      field_name[7] = "D3: "
      field_name[8] = "D4: "
   var rowcnt : n3

window 8, 40
touchscreen 13
for rowcnt = 1 to 8
           display rowcnt, 2, field_name[rowcnt]
           displayinput rowcnt, 10, customer_info[rowcnt],\
           "Enter ", field_name[rowcnt]
endfor
windowedit
for rowcnt = 1 to 8
   savechkinfo customer_info[rowcnt]
endfor
waitforclear
windowclose
touchscreen 3
endevent
```

See Also

ClearChkInfo command

SaveRefInfo

Description

This command is used to save the contents of an expression as part of tender reference information.

Syntax

SaveRefInfo *expression*[{*output_specifier*}][, *expression*\
[{*output_specifier*}]]...]

Argument	Description
expression	an *expression* that represents the information to be saved; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The **SaveRefInfo** command will only work with the **Event Tmed** procedure.
- Every time this command is used, a new reference line is created.
- Up to eight references may be saved with each tender and each may be up to 19 characters long. Text or fields greater than 19 characters will be truncated.

Example

The following script allows a user to enter delivery information on a To Go check:

```
event tmed : 10
    var rowcnt : n3
    var deliv_desc[6] : a15
    var deliv_info[6] : a20
    var cnt : n3

    deliv_desc[1] = "Name:"
    deliv_desc[2] = "Company:"
    deliv_desc[3] = "Address 1:"
    deliv_desc[4] = "Address 2:"
    deliv_desc[5] = "City:"
    deliv_desc[6] = "Phone:"

                                        //Display a window with address
                                        // prompts, and accept delivery
                                        // info from user

    window 6, 43
    for rowcnt = 1 to 6
        display rowcnt, 2, deliv_desc[rowcnt]
        displayinput rowcnt, 13, deliv_info[rowcnt], "Enter ", deliv_desc[rowcnt]
    endfor
    windoweditwithsave                          //Input can only be terminated
                                                // by cancel or save

    saverefinfo "DELIVER TO:"           //Save this line to check detail
    for cnt = 1 to 6                    //If user made an entry on a
                                        // line, save
                                        // the entry to check detail

        if len(deliv_info[cnt]) > 0 AND deliv_info[cnt] <> " "
            saverefinfo deliv_info[cnt]
        endif
    endfor
endevent
```

See Also

Event and **SaveRefInfox** commands

SaveRefInfox

Description

This command is used to save both the type and the contents of an expression as part of tender reference information. **SaveRefInfox** requires the reference type as an argument so that different types of reference detail can be distinguished if exported later.

Syntax

SaveRefInfox *ref_type*, *expression*[{*output_specifier*}][, *expression*\[{*output_specifier*}]]...]

Argument	Description
ref_type	a type N3 integer that represents the reference type; specify 0-255; 0 denotes no reference type, as with **SaveRefInfo**
expression	an *expression* that represents the information to be saved; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The **SaveRefInfox** command will only work with the **Event Tmed** procedure.
- Every time this command is used, a new reference line is created.
- Up to eight references may be saved with each tender and each may be up to 19 characters long. Text or fields greater than 19 characters will be truncated.

Example

The following script allows a user to enter delivery information on a To Go check:

```
event tmed : 10
    var rowcnt : n3
    var deliv_desc[6] : a15
    var deliv_info[6] : a20
    var cnt : n3

    deliv_desc[1] = "Name:"
    deliv_desc[2] = "Company:"
    deliv_desc[3] = "Address 1:"
    deliv_desc[4] = "Address 2:"
    deliv_desc[5] = "City:"
    deliv_desc[6] = "Phone:"
                                    //Display a window with address
                                    // prompts, and accept delivery
                                    // info from user
    window 6, 43
    for rowcnt = 1 to 6
        display rowcnt, 2, deliv_desc[rowcnt]
        displayinput rowcnt, 13, deliv_info[rowcnt], "Enter ",
deliv_desc[rowcnt]
    endfor
    windoweditwithsave              //Input can only be terminated
                                    // by cancel or save

    saverefinfox 15, "DELIVER TO:" //Save this line to check detail
    for cnt = 1 to 6        //If user made an entry on a
                            // line, save
                            // the entry to check detail

        if len(deliv_info[cnt]) > 0 AND deliv_info[cnt] <> " "
            saverefinfox 15, deliv_info[cnt]
        endif
    endfor
endevent
```

See Also

Event and **SaveRefInfo** commands

ScanBarcode

Description

This command is used to scan barcodes that contain data larger than 40 characters (e.g., QR barcodes).

Syntax

ScanBarcode returned data, message

Remarks

- The message "**Scan your Barcode**" will be displayed in the "Yellow Object Bar" (YOB) on the Workstation display to prompt the user to perform the operation.
- No Alpha Keyboard entry screen will be displayed.

Example

This example will scan a barcode and save it to a file.

```
event inq:1
var couponCode : A512
  var fn :N2
scanbarcode couponCode ,"Scan your Barcode"
fopen fn,"\store\posclient\sim\barcode.txt", append , local
  if fn = 0
      errormessage "File already open or missing:
\POSClient\SIM\barcode.txt"    // @FILE_ERRSTR
      exitcontinue
   else
      fwrite fn, 1, couponcode
      fclose fn
      errormessage "Barcode saved to file"
      exitCancel
 endif

endevent
```

SetIslTsKey

Description

This command defines a key to be displayed on a touchscreen, allowing one to define a key that normally would be programmed in the *Touchscreens* module.

Syntax

SetIslTsKey *row*, *col*, *num_rows*, *num_cols, font*, \
key_expression, *expression*

Argument	Description
row	integer expression defining row coordinate of key (1-6)
col	integer expression defining column coordinate of key (1-10)
num_rows	integer expression defining the key height in rows (1-6)
num_cols	integer expression defining the key width in columns (1-10)
font	font size integer expression (1-3)
key_expression	key value to be returned when this key is pressed; it may be one of the following: **Key** function *user_variable* (type Key only) *system_variable* (@KEYS only)
expression	descriptor to appear on the key as it is displayed; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- Up to nine keys may be defined with the **SetIslTsKey**.
- Any previously defined touchscreen keys are automatically cleared each time a script executes. However, if two or more touchscreens are defined within an event, the **ClearIslTs** command must be used to clear the touchscreen keys.

Example

The subroutine below first clears any previously defined touchscreen keys and displays two touchscreen keys, [YES] and [NO], which are defined by the **SetIslTsKey** command. This subroutine displays these keys as the operator is issued a prompt by the system and captures the operator's input.

```
sub get_yes_or_no( ref answer, var prompt_s:A38 )
   var keypress : key
   var data : A20

   clearislts
   setisltskey 2, 2, 4, 4, 3, @KEY_ENTER, "YES"
   setisltskey 2, 6, 4, 4, 3, @KEY_CLEAR, "NO"
   displayislts

   inputkey keypress, data, prompt_s
   if keypress = @KEY_ENTER
      answer = 1
   else
      answer = 0
   endif
endsub
```

See Also

ClearIslTs, **DisplayIslTs**, and **PopUpIslTs** commands

SetReRead

Description

This command allows OPS to re-read the ISL script for new or changed ISL scripts.

Syntax

SetReRead

Remarks

- Previously, OPS would always check if an ISL script had changed before processing the event. Now that many more events have been added, continuously checking file status would be an unnecessary strain on system resources, especially since this feature is used only for debugging scripts.
- The ISL script will also be reread if `/micros/simphony/etc/isl.reread` is present when POS Operations is started.

SetSignOn[Left/Right]

Description

These commands determine where ISL places the minus sign when formatting numbers.

Syntax

SetSignOnLeft

or

SetSignOnRight

Remarks

- This is a global command.
- By default, ISL puts the minus sign to the ***right*** of the number being displayed (for example, -45 will display as "45-"). If this is not acceptable, then executing the **SetSignOnLeft** command will cause ISL to format ***all*** numbers with the minus sign on the left.

  REMEMBER: Using the SetSignOnRight or SetSignOnLeft command does NOT change the way ISL reads input data. Any external data read and interpreted by ISL, such as received messages, file read operations, and operator input, must have the negative sign on the left-hand sign of the value.

  For example, if the ISL script prompted for an amount entry, and the operator entered *1.23-*, the value would not be accepted as positive. Any negative PMS entries must have the sign on the left side of the field.

Example

```
event inq : 1

   waitforclear -123            //will display '123-' by default
   setsignonleft
   waitforclear -123            //will display '-123'
   setsignonright
   waitforclear -123            //will display '123-'

endevent
```

SetString

Description

This command will replace all, or a specific number of, characters in a string with a particular character.

Syntax

SetString *main_string*, *character_string*[, *count*]

Argument	Description
main_string	the *string* in which characters will be replaced; it can be any *string_variable*
character_string	a *string* whose first character will be used to replace characters in *main_string*; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
count	optional *count* of characters to set the *main_string* to. If not specified, the entire *main_string* will be replaced with the first character of the *character_string*

Example

The following script constructs and prints a date string bordered by hash marks at the workstation's check printer:

```
sub print_date
    var date : a9
    var hash_mark : a24              //Use format to build a date
                                     // string
    format date as @DAY, "-", month_arr[@MONTH], "-", @YEAR

    setstring hash_mark, "="
    startprint @CHK            //Print what follows at this UWS
                               // check printer
        printline hash_mark    //Print the date string
        printline @DWON, @REDON, date{=16}// double-wide, in red
        printline hash_mark    // and centered between
                               // hash marks
    endprint
endsub
```

SimDB

Description

This command is used by the SIM to send a request to the SIMDB DLL and then receive a response.

Syntax

SimDB *interface_number, request_msg, response_message*

Argument	Description
interface_number	the interface definition object number
request_msg	the request message
response_message	the response message

Remarks

If the SIM script is linked directly to the SIMDB interface, then @PMSNUMBER should be used. If the SIM script is linked to a standard PMS interface, then the object number of the SIMDB interface must be specified. If the standard PMS interface has a value in the **SIMDB Link** field in *Interfaces* | *General tab* within the Enterprise Management Console (EMC), then the @SIMBLINK system variable can be used.

Example 1

// Use the PMS number linked to this script

```
Simdb @pmsnumber, req, resp
```

// Use PMS number 15

```
Simdb 15, req, resp
```

// Use the SIMDB link from the PMS linked to this script

```
Simdb @simdlink, req, rsp
```

- `request_msg` is the string to be sent to the SIMDB DLL.
- `response_msg` is the string which will be filled with the response from the SIMDB DLL request.

Example 2

```
event inq:1

       var req:A80
       var rsp:A80

       req = "name|fred"
       simdb @pmsnumber, req, rsp
       waitforclear "rsp=", rsp

endevent
```

See Also

@PMSNUMBER and **@SIMDBLINK** variables

Split

Description

This command is used to split a field-separated string into separate variables.

Syntax

Split *string_to_split*, *field_sep_char*, *user_variable* or *list_spec* \
[, *user_variable* or *list_spec*...]

Argument	Description
string_to_split	the field-separated *string* to split
field_sep_char	the character used to the separate fields in the *string_to_split*; use the **Chr** function to define the character required
user_variable	a *user_variable* which will be assigned one of the individual fields from the *string_to_split*
list_spec is defined as: *number_records*	a user integer variable containing the number of records to be built from the *string_to_split*
field_array[: *field_array*]	an *array_variable* that will hold one field per record; a field can be split into more than one *array* by separating the *array_variables* using a colon (:)

Remarks

If the system variable @STRICT_ARGS is set to 1, ISL will ensure that the variable count in the **Split** command matches the number of fields in the file record. If an incorrect number of fields is specified in the statement or the file is corrupted, an error message will be generated.

Example

The following example will assume the PMS has sent the guest name and room number as one field. The name is separated by the room number with the [fs] character. Without the **Split** command, the script would have to search through the field one character at a time until the [fs] character was found. The **Split** command, however, will automatically split the data into fields:

```
event rxmsg : guest

   var name_and _room : a25
   var name : a20
   var room_number : n5

                                  //Get the data as one field, and
                                  //split it into 2 fields

   name_and_room, "[fs]", name, room_number

                                  //Display data in window
   window 5, 30, "Guest Inquiry"
   display 2, @CENTER, name
   display 4, @CENTER, room_number
   waitforclear
endevent
```

See Also

Format and **SplitQ** commands; **Chr** function

SplitQ

Description

This command is used to split a field-separated string into separate variables and strips the quotes from the quoted string.

Syntax

SplitQ *string_to_split*, *field_sep_char*, *user_variable* or *list_spec* \
[, *user_variable* or *list_spec*...]

Argument	Description
string_to_split	the field-separated *string* to split
field_sep_char	the character used to the separate fields in the *string_to_split*; use the **Chr** function to define the character required
user_variable	a *user_variable* which will be assigned one of the individual fields from the *string_to_split*
list_spec is defined as: *number_records*	a user integer variable containing the number of records to be built from the *string_to_split*
field_array[: *field_array*]	an *array_variable* that will hold one field per record; a field can be split into more than one *array* by separating the *array_variables* using a colon (:)

Remarks

- The **SplitQ** command operates in the same way as the **Split** command, except that it will strip quotes from quoted strings. The **SplitQ** command is generally used to split lines from a file, where strings are usually quoted.

- If the system variable @STRICT_ARGS is set to 1, ISL will ensure that the variable count in the **SplitQ** command matches the number of fields in the file record. If an incorrect number of fields is specified in the statement or the file is corrupted, an error message will be generated.

See Also

Format and **Split** commands; **Chr** function

StartPrint...EndPrint[FF/NOFF]

Description

These commands are used to start and end a print session on any MICROS printer (as opposed to a line printer). The **StartPrint** command is used to select the printer and start the print session and the **EndPrint** command ends the print session. The Form Feed [**FF**] and No Form Feed [**NOFF**] may also be used with the **EndPrint** command, depending on the needs of the application and printer default.

Syntax

StartPrint *printer_name*|*expression* [, *backup_dten* [, *reference_line*]]
.
.
.

EndPrint [or **EndPrintFF** or **EndPrintNOFF**]

Argument	Description
printer_name	the object number of the printer
expression	any *variable* that contains the object number of a printer in the system
backup_dten	the object number for the backup printer that overrides the backup printer programmed in the database
reference_line	the text string to be displayed in the printer error window if an error occurs during printing

Remarks

- The *expression* is defined using one of the ISL Printer system variables in the table below.

ISL Printer Name	Printer Assignement in Workstations | Printers	Valid Printer Types	# of char SW	# of char DW	Red	FF Default	Extended Print
@RCPT	Customer Receipt	Roll	32	16	Optional	Yes	Yes
@CHK	Guest Check	Roll	32	16	Optional	Yes	Yes

ISL Printer Name	Printer Assignement in Workstations | Printers	Valid Printer Types	# of char SW	# of char DW	Red	FF Default	Extended Print
@ORDR# (where # is 1 - 15 referring to the appropriate order device)	Order (local or remote)	Roll	32	16	Optional	Yes	Yes
		KDS	19	No	No	No	No
@VALD	Validation	Roll	32	16	Optional	Yes	Yes

- The physical printer is selected based on the workstation printer assignments in the Simphony database. For example, when @CHK is selected, the ISL will pass along the information to the Check Printer defined for the workstation, from which the Event was initiated.

- In the case where a printer fails and a backup printer is programmed, the printing session will notify the operator at the workstation that the printing session has failed and the print session has been sent to the backup printer.

- Printing will not begin until the **EndPrint** command is executed. Each printer will FormFeed or not, depending on its default characteristic (see table); **EndPrint** will follow the printer's default. Use either **EndPrintFF** or **EndPrintNOFF** to force the opposite action.

- If single-wide printing is used, anything over 32 characters will be truncated. If the @DWON Print Directive is used, anything over 16 characters will be truncated.

- If the @REDON Print Directive is used with the @KDS device, the output will be Bright. If the @REDON Print Directive is used with a printer, the correct type of ribbon is required (black/red) for output to print in red.

- Note that only one print session may be active at any one time from the same workstation.

POS Setup

To enable the printer, enter the Simphony database and verify that the printer:

- is entered as a device in the *Workstations* module

- has its object number entered as one of the printers in the *Workstations* module

Example

The following script allows a user to enter a 13 line message and print it at the kitchen printer:

```
event inq : 1

    var kitchen_msg[13] : a20
    var sender_name : a20
    var rowcnt : n3
    var hash_mark : a24
    window 14, 22                                   //Display the window

    displayinput 1, 2, sender_name, "Enter your name"
                                                    //Accept users name
                                                    //Have user input the message

    for rowcnt = 1 to 13
        displayinput rowcnt + 1, 2, kitchen_msg[rowcnt], "Enter kitchen message"
    endfor
    windowedit                                      //Only save or cancel will

    setstring hash_mark, "="                        // end input
    startprint @ordr1                               //Start the print job at
                                                    // remote printer1

    printline hash_mark

        printline "Message from ", sender_name
        printline hash_mark
        for rowcnt = 1 to 13
            if len(kitchen_msg[rowcnt]) > ""
                printline kitchen_msg[rowcnt]
            endif
        endfor
        printline "======= END MESSAGE ========"
    endprint
endevent
```

See Also

- **PrintLine** command
- "ISL Printing"

Sub... EndSub

Description

These commands are used to declare the start and end of a subroutine. A subroutine may be used by an event or another subroutine by using the **Call** command.

Syntax

Sub *name* [(**Ref** | **Var** *parameter* [, **Ref** | **Var** *parameter*]...)]
.
.
.
EndSub

Argument	Description
name	the subroutine name
Ref	ISL keyword
Var	Var command, see page 7-184
parameter	a *variable* or *expression* passed from the associated **Call** command; it may be one of the following: *variable* by reference *array* by reference *expression* by value, i.e., *local_variable_name* : [A, $, N] Size, or **Key**

Remarks

- Each **Sub** command defines the number and type of parameters that can be passed in. If the **Call** command has the incorrect number of arguments or the incorrect type of arguments, an error will display.
- **Sub** commands are not allowed within an event.

```
event inq : 1
    sub                          // will display error
       .
       .
       .
    endsub
endevent
```

- Each subroutine has access to the calling event's variables, all global variables, and may declare their own local variables.
- The *name* may be any length up to 255 characters and must begin with a letter A - Z, a - z, or the underscore character (_). It may include any character in the range A - Z, 0 - 9, and the underscore character.

Using Subroutines

ISL subroutines can be passed parameters in the same way as C, BASIC, or Pascal subroutines. Data can be passed in by two methods: value or reference.

By Value

- A parameter in a **Sub** command is considered to be by value if declared as a normal ISL variable. Example:

```
sub get_name( var target_name:A20, var target_id:N5 )
```

Both `target_name` and `target_id` are local variables which are assigned the value passed in with the **Call** command. A **Call** command to call this subroutine could be:

```
call get_name( "Smith", 145 )
```

Any type of ISL *expression* can be passed in. Example:

```
call get_name( "Smith", 145 + offset )
call get_name( user_input, elist[ 25 ] )
```

- The *expression* type (string, numeric,...) can be different from the one declared in the **Sub** command. Example:

```
call get_name( "Smith", "45" )
```

- The assignment of the expression to the subroutine parameter follows the same rules when setting a variable in a normal assignment expression.

- Arrays cannot be passed in by value.

- When a variable is passed in by value (using the **Var** command in the **Sub** statement), a copy is made of the variable and given the name specified in the **Sub** statement. Any change made to a variable passed by value in a subroutine does ***not*** affect the original value of the variable.

```
event inq : 1
    var i : n5 = 10                     //Set i to 10
    call mysub ( i )                    //Pass in i by value
    waitforclear "i = ", i              //i is still equal
                                        // to 10
endevent

sub mysub ( var j : n5 )
    j = 20                              //Change local copy,
                                        // not the original
endsub
```

By Reference

- There are two types of data that can be passed in by reference: variables and arrays. To pass either, the **Ref** variable is used in the **Sub** statement. No type information is specified for the referenced variable.

For example, in the following line `status` is passed by reference, and `prompt_string` is passed by value.

```
sub mysub( ref status, var prompt_string:A20 )
```

The following example shows correct and incorrect ways to invoke `mysub`:

```
var result:N5
    call mysub( result, "Enter data" )    // Correct
    call mysub( result+1, "Enter data" )

              // Incorrect: 'result+1' is not a variable.

    call mysub( (result), "Enter data" )
              // Incorrect: '(result)' is considered an
              // expression
```

- To pass an array, empty brackets must be placed after the array name both in the **Sub** and the **Call** commands. Example:

```
sub mysub( ref data[], var prompt_string:A30 )
```

The following example passes an array to `mysub`:

```
var array[ 10 ]:A20
    call mysub( array[], "Enter data" )      // Correct
    call mysub( array, "Enter data" )
              // Incorrect. Need [] after array
```

- When a variable is passed in by reference, any change in the subroutine to the variable affects the original value of the variable. The name after the **Ref** variable can be thought of as being another name for the variable passed in.

```
event inq : 1
    var i : n5 = 10                      //Set i to 10
    call mysub ( i )                     //Pass in i by
                                         //reference
    waitforclear "i = ", i               //i is now equal
                                         //to 20
endevent

sub mysub ( ref j )
    j = 20                               //Change original
                                         //value of i and j
                                         //to 20
endsub
```

Example

The event below calls a subroutine, format_data, that formats the current date as follows: dd-mmm-yyyy.

```
event inq : 1
var date : a11

    call format_date( date, @day, @month, @year)
    waitforclear date
endevent

sub format_date( ref date, var day : n5, var month : n5, var year : n5 )
    var month_arr[12] : a3
    month_arr[1] = "JAN"
    month_arr[2] = "FEB"
    month_arr[3] = "MAR"
    month_arr[4] = "APR"
    month_arr[5] = "MAY"
    month_arr[6] = "JUN"
    month_arr[7] = "JUL"
    month_arr[8] = "AUG"
    month_arr[9] = "SEP"
    month_arr[10] = "OCT"
    month_arr[11] = "NOV"
    month_arr[12] = "DEC"

    format date as day, "-", month_arr[ month ], "-", year
endsub
```

See Also

Call and **Var** commands

System

Description

This command allows the user to execute a Windows batch or executable file.

Syntax

System *command*[{*output_specifier*}]

Argument	Description
command	a properly formatted Windows batch or executable file, surrounded by quotes, or a *string_variable* that contains a command
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

The *command* may be a comma-separated series of *strings* and *string_variables*.

Forward slashes (/) should be used instead of Windows backslashes (\) when specifying a path.

If a directory is not specified, SIM will default to *MICROS\Simphony\Etc*. Any command that is not in the user's PATH or in the *Etc* directory should be relative to *Etc*. Refer to the examples on page 7-171.

Example 1

This is a valid example because the *bootptab* file is in the *Etc* directory.

```
event inq : 10
    var scriptname : a40 = "script.save"
    system "cp bootptab",scriptname
endevent
```

Example 2

Since the *Export* directory is not in the user's PATH, the user was required to specify the location of *script.sh* relative to *Etc*.

```
event inq : 10
    var scriptname : a40 = "script.save"
    system "cp ../export/script.sh ..export/",scriptname
endevent
```

Example 3

If commands are in the user's PATH, then the location of the executable file is not needed.

In this example, Notepad® will be launched on the server. Although Notepad® is not in the default directory, it is located in a directory which is defined in the user's PATH.

```
event inq : 10
   var scriptname : a40 = "notepad.exe"
   system scriptname
endevent
```

Note: *These examples are provided to give a general idea of how the* ***System*** *command works. They do not provide all of its possibilities or limitations.*

Touchscreen

Description

This command allows the system to display a pop-up touchscreen. This feature improves operator prompting, making the system easier to use.

Syntax

Touchscreen *numeric_expression*

Argument	Description
numeric_expression	the object number of a pre-defined touchscreen, programmed in the Simphony database

Remarks

- The command references a touchscreen within the *Touchscreen* module. This Touchscreen number must already be setup within the Simphony database.
- The Touchscreen will disappear when this interface session is complete. As an example, when prompting for the guest's room number, the Touchscreen could display a numeric keypad. Touchscreen 0 will cause the default touchscreen to be used.
- This command is used only with the WS4(+) and the PCWS.

TxMsg

Description

This command defines the applications data segment of a message that will be transmitted over the interface.

Syntax

TxMsg *expression*[{*output_specifier*}][, *expression*[{*output_specifier*}] \...]

Argument	Description
expression	an *expression* that will be transmitted over the interface; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The **TxMsg** command must be followed by the **WaitForRxMsg** command.
- If more than one field is required, multiple *expressions* must be separated by commas. These commas will be replaced in the message received by the interfaced system by field separator characters (ASCII 1CH).
- A statement may continue over multiple lines by including the line continuation character (\) at the end of each line. The line continuation character is very useful as **TxMsg** commands tend to be very lengthy.

Example

A transmission message that includes the check employee number, the check number, tendered total, and a declared room number field could be defined as follows:

```
event inq : 1
   var room_num : a4
   input room_num, "Enter Room Number"
   txmsg "charge_inq",@CHKEMP,@CHKNUM,@TNDTTL,room_num
                                        //The first field (charge_inq) is
an
                                        // example of an identifying
string
                                        // that the PMS might use to
process
                                        // messsage from the POS.
   waitforrxmsg
endevent
event rxmsg : charge_declined           //This is one of the PMS response
                                        // possibilities
   var room_num : a4
   rxmsg room_num
   exitwitherror "Charge for room ", room_num," declined"
endevent
```

See Also

Event, **RxMsg**, **TxMsgOnly**, and **WaitForRxMsg** commands

TxMsgOnly

Description

This command sends a message to a PMS without waiting for a response.

Syntax

TxMsgOnly *expression*[{*output_specifier*}][, *expression*[{*output_specifier*}] \...]

Argument	Description
expression	an *expression* that will be transmitted over the interface; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Example

A transmission message that includes the check employee number, the check number, tendered total, and a declared room number field could be defined as follows:

```
event inq : 1
    var room_num : a4
    input room_num, "Enter Room Number"
    txmsgonly "charge_post",@CHKEMP,@CHKNUM,@TNDTTL,room_num

    //No message will be received. If the PMS responds, it will
   // be ignored.

endevent
```

See Also

TxMsg command

UpperCase

Description

This command is used to convert a string variable to uppercase.

Syntax

UpperCase *string_variable*

Argument	Description
string_variable	the string that will be changed to an uppercase *user_variable* (string)

See Also

LowerCase command

UseBackupTender

Description

This command instructs the ISL to switch to the programmed backup Tender.

Syntax

UseBackupTender

Remarks

- Tender Media PMS Option **Switch to Alternate Tenders if PMS Timeout** must be enabled. If backup tender is not programmed, this command has no effect.
- This is useful for posting to other tenders when the primary PMS is not active.

Example

```
event rxmsg : _Timeout
   waitforconfirm "Post to backup tender"
   usebackuptender
   exitcontinue
endevent
```

See Also

Event Tmed command

Use[Compat/ISL]Format

Description

These commands are used to instruct ISL to use the ISL message format or the Simphony-standard message format.

Syntax

UseCompatFormat

or

UseISLFormat

Remarks

- These commands are global.
- If the Simphony-standard message format is selected, the Application Data Segment will not contain the two-byte sequence number or the retransmission flag, or the FS after the STX. For more information, please see “Message Formats” on page 2-3.

Use[ISL/STD]TimeOuts

Description

These commands instruct ISL how to process a PMS timeout. **UseISLTimeOuts** will search the script file for an **RxMsg** Event with an Event ID of **_Timeout**. **UseSTDTimeOuts** will use the standard Simphony error messaging for a PMS timeout.

Syntax

UseISLTimeOuts

or

UseSTDTimeOuts

Remarks

- The Event ID **_Timeout** will bypass the standard Simphony error messaging for a PMS timeout and process instructions from the ISL script file.
- For example, if the **UseISLTimeOuts** command is used and the PMS does not respond to an ISL message within the timeout period, the ISL script may then ask the user if the processing should be cancelled, or if a backup tender should be used.

See Also

Event, **TxMsg**, and **RxMsg** commands

UseSortedDetail

Description

This command causes detail system variables to access consolidated detail.

Syntax

UseSortedDetail

Remarks

This command applies to the detail system variables only.

Example

The following example, which assumes that check information lines are equal to one, displays the name, tax amount, and quantity for each line of detail. Use this event to see how **UseSortedDetail** changes the detail output format for consolidated menu items.

```
event inq : 1
    var chk_line = 1
    UseSortedDetail
    while chk_line <= @numdtlt
        call display_dtl
        chk_line = chk_line + 1
    endwhile
endevent

sub display_dtl
    window 5,40
    display 2,5, "Menu Item Name         = " , @dtl_name[chk_line]
    display 3,5, "Tax Amount             = " , @dtl_taxttl[chk_line]
    display 4,5, "Quantity               = " , @dtl_qty[chk_line]
    if @detailsorted = 1
    display 5,5 "Detail Is Consolidated"
    else
    display 5,5 "Detail Is Not Consolidated"
    endif
    waitforclear
endsub
```

See Also

UseStdDetail command and @DETAILSORTED system variable

UseStdDetail

Description

This command causes the detail system variables to access raw detail.

Syntax

UseStdDetail

Remarks

This command applies to the detail system variables only.

See Also

UseSortedDetail command and @DETAILSORTED system variable

UseTMSFormat

Description

This command is used to format messages using the TMS message format.

Syntax

UseTMSFormat

Remarks

This command is global.

Var

Description

This command is used to declare user variables. When it is used outside an event or subroutine, the user variable will be accessible globally. When it is used inside an event or subroutine, the variable is considered "local" and is only allowed to be used by that event or any subroutine called by the event.

Syntax

Var *user_variable* : *variable_specifier*

Argument	Description
user_variable	a user-defined name for a variable. If an *array_variable* is being declared, the *user_variable* must be followed immediately by brackets containing the maximum number of items the *array_variable* may contain, e.g., name[100]
variable_specifier	made up of a type and a size specifier. The *type_specifier* can be an N, $, or A. The size specifier can be either a number (i.e., 12) or an integer *expression* enclosed in parentheses, i.e., 12 + 15). If the **Key** *type_specifier* is used, the size should not be included. The *variable_specifier* is concatenated together, and defines whether the *variable* is alphanumeric, numeric (integer), a decimal, or a key, and, if its one of the first three, how many characters/ digits it contains

Example

```
var guest_name :  A20

var guests[max_guests] :  A(max_name_length)
```

Remarks

- The *user_variable* name can be up to 255 characters in length and must begin with a letter A - Z, a - z, or the underscore character (_). It may include any character in the range A - Z, 0 - 9, and the underscore character. Case is insignificant and the name must not contain spaces.

- It is possible to declare a variable with a size that is defined by an *expression*, rather than a hard-coded number. These are known as variable-size variables. If the ISL encounters a left parentheses immediately following the type of the variable, it will assume that an *expression* follows, which defines the variable's size. Variable-size variables can be declared as follows:

```
var i : n3 = 15
var window_width : n(i + 15)
```

  If `i = 15`, then the line shown above would be the same as:

```
var window_width : N30
```

- If the *size* of the *variable* is not known when the variable is being defined, then the variable size may be placed within parentheses.

```
var string : A10
var string : A( 7 + 3 )                    //Same as A10
var string : A( num_guests * 2 )           //Depends on value
                                           // num_guests
```

- The *type_specifier* is different for each of the four types of user-defined variables. Note that no size is specified when declaring a key variable.

Field Type	Type Specifier	Example
Numeric	n or N	n3
Decimal	$	$6
Alphanumeric	a or A	a25
Key	key	key

For more detail about *variable_specifiers*, refer to “Data Types” on page 4-3.

The declaration of a four digit room number would be defined as:

```
var room_num:N4
```

An array with six elements may also be defined as:

```
var message_text[6]:A32
```

More than one variable can be declared on the same line, as long as they are separated by commas:

```
var room_num : a5, guest_count : n3
```

Variables can be declared and defined on the same line:

```
var city_name : a10 = “Charleston”
```

- Zero-length arrays and zero-size variables are allowed.
- All variables are cleared when a new Event is executed, unless the **RetainGlobalVar** command is used (which only affects global variables).

See Also

[Retain/Discard]GlobalVar command

WaitForClear

Description

This command requires the operator to press the [Clear] key before proceeding. It is often used after the **Display** command.

Syntax

WaitForClear [*prompt_expression*[{*output_specifier*}]]\
[, *prompt_expression*[{*output_specifier*}]...]

Argument	Description
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The default prompt, “Press Clear to continue”, will appear on the prompt line of the workstation. This may be overridden by providing a *prompt_expression* with the command.
- The combined length of all *prompt_expressions* must not exceed 38 characters (including spaces); extra characters will be truncated.

Example

The following script would display some text, then require the operator to press the [Clear] key before continuing:

```
event inq : 1
   window 1, 14
   display 1, 2, "Hello"          //"You say Hello"
   waitforclear
   display 1, 2, "Goodbye"        // "and I say Goodbye"
   waitforclear
endevent
```

See Also

Display, **WaitForConfirm**, **WaitForEnter**, and **WaitForRxMsg** commands

WaitForConfirm

Description

This command requires the operator to press the [Enter] key or the [Clear] key.

Syntax

WaitForConfirm [*prompt_expression*[{*output_specifier*}]]\
[, *prompt_expression*[{*output_specifier*}]...]

Argument	Description
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- If the operator presses the [Enter] key, the script will continue.
- If the operator presses the [Clear] key, the operation will be cancelled and the script will be exited. The default prompt, “Press [Enter] to continue”, will appear on the prompt line of the workstation. This may be overridden by providing a *prompt_expression* with the command.
- The combined length of all *prompt_expressions* must not exceed 38 characters (including spaces); extra characters will be truncated.

Example

The following script is a transmission message that includes the check employee number, the check number, tendered total, and a declared room number field. After the `room_num` is displayed, the script will wait for the [Enter] key or the [Clear] key:

```
event inq : 1
    var room_num : a4
    input room_num, "Enter Room Number"
    waitforconfirm
    txmsg "charge_inq",@CHKEMP,@CHKNUM,@TNDTTL,room_num
                                                    //The first field (charge_inq)
is an
                                                    // example of an identifying

string
                                                    // that the POS might use to

process
                                                    // message from the POS.
    waitforrxmsg
endevent

event rxmsg : charge_declined          //This is one of the PMS response
                                       // possibilities
    var room_num : a4
    rxmsg room_num
    exitwitherror "Charge for room ", room_num," declined"
endevent
```

See Also

Display, **WaitForClear**, **WaitForEnter**, and **WaitForRxMsg** commands

WaitForEnter

Description

This command requires the operator to press the [Enter] key before proceeding.

Syntax

WaitForEnter [*prompt_expression*[{*output_specifier*}]]\
[, *prompt_expression*[{*output_specifier*}]...]

Argument	Description
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The default prompt, "Press Enter to continue," will appear on the prompt line of the workstation. This may be overridden by providing a *prompt_expression* with the command.
- The combined length of all *prompt_expressions* must not exceed 38 characters (including spaces); extra characters will be truncated.

Example

The following script would display the guest's name and require the operator to press the [Enter] key before continuing:

```
event rxmsg : guest
    var name : a20
    rxmsg name
    waitforenter "guest is ", name
endevent
```

See Also

Display, **WaitForClear**, **WaitForConfirm**, and **WaitForRxMsg** commands

WaitForRxMsg

Description

This command is used after a message has been transmitted over the interface so that the system waits for a response.

Syntax

WaitForRxMsg [*prompt_expression*[{*output_specifier*}]]\
[, *prompt_expression*[{*output_specifier*}]...]

Argument	Description
prompt_expression	an *expression* displayed on the prompt line, usually to instruct the user what to enter; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The default prompt, “Please Wait--Sending Message,” will appear on the prompt line of the workstation. The default may be overridden by providing a *prompt_expression* with the command.
- The **WaitForRxMsg** is not a stand-alone command; the **TxMsg** command must precede the **WaitForRxMsg**. The **WaitForRxMsg** command is an implicit return.
- When a **TxMsg** statement is followed by a **WaitForRxMsg** statement and a response message is received from the interfaced system, it will assume that there is a return event (**Event Rxmsg**) that corresponds to the message from the interfaced system.

Example

A transmission message that includes the employee check number, the check number, tendered total, and a declared room number field could be defined as follows:

```
event inq : 1
    var room : n5
    input room, "Room? "
    txmsg "room_inq", room
    waitforrxmsg                          //Script stops here
    waitforclear                          //Not executed because
                                          // waitforrxmsg has
                                          // terminated event
endevent
```

See Also

Event RxMsg, **RxMsg**, **TxMsg**, **WaitForClear**, **WaitForConfirm**, and **WaitForEnter** commands

While...EndWhile

Description

The **While** command is used to implement a loop structure. The **EndWhile** is used to end the loop.

Syntax

While *expression*

.
.
.

EndWhile

Argument	Description
expression	the loop condition *expression* to be evaluated; it may be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*

Remarks

- When ISL encounters a **While** statement, it will execute all statements within the **While** and its corresponding **EndWhile** command until the expression in the **While** command becomes FALSE. If the expression is not initially true, then the statements within the **While** block are not executed.

```
while not feof( fn )
    call process_data( fn )
       .
       .
       .
endwhile
```

  The **While** example shown above is the standard method of using the **Feof** function to test for the end of the file being processed.

```
endsub
```

- The **While** command can be nested within other **While** commands.

- The expression in the **While** command is similar to the conditional expression within the **If** command.

Example

```
event inq : 1
    while data_ok = 1
        max_data = 10
        while i < max_data
            call get_next_line
            i = i + 1
        endwhile
    endwhile
endevent
```

See Also

For, **ForEver,** and **If** commands; **Feof** function

Window

Description

This command will draw a window on the operator display and is required in order to display information referenced by the various **Display** commands.

Syntax

Window *row, column*[*, expression*[{*output_specifier*}]...]

Argument	Description
row	the number of rows the window should contain; valid entries are 1 to 14
column	the number of columns the window should contain; valid entries are 1 to 78
expression	an *expression* that represents the title of the window, which will appear centered in the top line of the window itself (above the first row); it may be on of the following: *user_variable* *string*
{*output_specifier*}	one or more of the *output_specifiers* that determine the format of the output fields; see full definition beginning on page 7-10

Remarks

- The *row* and *column* values in the window are a function of how much the programmer needs to display. Refer to the **Display** commands to determine these requirements.
- The optional window title may be a text string or a combination of *user_variables* and *string_variables*.
- The maximum number of characters that will be displayed in the Window title is the number of *columns* minus 1; extra characters will be truncated.

Example

The following script will read the information from Track 1 of a credit card and display it in a window:

```
event inq : 1
    var cardholder_name: a26
    var account_num: n19
    var expiration_date: n4
    var track1_data: a79
    window 3, 78
    displaymsinput 1, 2, cardholder_name{m1, 2, 1, *}, "Enter Guest Name", \
                2, 2, account_num{m1, 1, 1, *}, "Enter Account Number", \
                3, 2, expiration_date{m1, 3, 1, 4}, "Enter Expiration"
    windowinput
    waitforclear
endevent
```

See Also

Display, **DisplayInput**, **DisplayMSInput**, **WaitForClear**, **WaitForConfirm**, **WaitForEnter**, **WindowClear**, and **WindowClose** commands

WindowClear

Description

This command clears the contents of a currently displayed window.

Syntax

WindowClear

Example

The following script builds a window, displays some text, clears the text from the window (after the operator presses [Clear]), then displays some more text:

```
event inq : 1
   window 1, 14
   display 1, 2, "Hello"                //"You say Hello"
   waitforclear
   windowclear
   display 1, 2, "Goodbye"              // "and I say Goodbye"
   waitforclear
endevent
```

See Also

Window command

WindowClose

Description

This command closes a currently displayed window.

Syntax

WindowClose

Remarks

Windows close automatically when a script is exited, or when a new window is built with the **Window** command. The **WindowClose** command allows the script writer to close a window before either of these events has occurred.

Example

The following script builds a window, displays some text, then closes the window (after the operator presses [Clear]):

```
event inq : 1
   window 1, 14
   display 1, 2, "Hello"                //"You say Hello"
   waitforclear
   windowclose
   display 1, 2, "Goodbye"              // "and I say Goodbye"
   waitforclear
endevent
```

See Also

Window command

Window[Edit/Input][WithSave]

Description

One of these commands is required any time the **DisplayInput** and **DisplayMSInput** commands are used. The **WindowEdit** and **WindowInput** determine whether or not the contents of the variables are cleared before being displayed. The [**WithSave**] option, with either command, determines how the input session will be terminated.

Syntax 1

WindowEdit[**WithSave**]

Syntax 2

WindowInput[**WithSave**]

Remarks

- When one of the **WindowEdit** or **WindowInput** commands is found, the SIM will look for the first and subsequent **DisplayInput** or **DisplayMSInput**, since the window was drawn and executes it. Therefore, the **WindowEdit** and **WindowInput** command must be preceded by both a **Window** command and at least one **DisplayInput** or **DisplayMSInput** command.

- The **WindowEdit** and **WindowEditWithSave** commands do not clear the variables. If the *input_variables* contain data, the contents will be displayed in the input window. These commands are useful when the operator must change only a few fields or characters of a record. The **WindowInput** and **WindowInputWithSave** commands clear the variables before they are displayed. All previous information in the fields is lost after the command is executed. These commands are useful for entering new information.

- **WindowEdit\Input** can be used with the **Display[MS]Input** commands to build a screen of input fields in order to accept input from the user. Navigating among the input fields is achieved with the movement keys: up arrow, down arrow, home, and end. [Enter] can also be used to navigate, which moves the focus to the next field, and [Clear], which moves the focus to the previous field.

- The **WindowInput** and **WindowEdit** commands complete the Input session when the cursor is on the last field and the user presses the [Enter] key or the [Down Arrow] key. The Input session is cancelled if the cursor is on the:
    - first field and the user presses the [Clear] key or the [Up Arrow] key.
    - last field and the user presses [Enter] or [Down Arrow].
- The **WindowInputWithSave** and **WindowEditWithSave** commands require the operator to either press the [Save] key to complete the Input session, or press the [Cancel] key to cancel it. If the cursor is on the last edit field and either the [Enter] key or the [Down Arrow] key is pressed, the cursor will roll to the first field. Likewise, if the cursor is on the first edit field and either the [Clear] key or the [Up Arrow] key is pressed, the cursor will roll to the last field.

Example 1

The following script will read the information from Track 1 of a credit card:

```
event inq : 1
   var cardholder_name: a26
   var account_num: n19
   var expiration_date: n4
   var track1_data: a79

   window 3, 78
   displaymsinput 1, 2, cardholder_name{m1, 2, 1, *}, "Enter Guest Name", \
    2, 2, account_num{m1, 1, 1, *}, "Enter Account Number", \
    3, 2, expiration_date{m1, 3, 1, 4}, "Enter Expiration"
   windowinput
   waitforclear
endevent
```

Example 2

The following script will allow input of customer information in a window:

```
event inq : 1
    var name                  : a20
    var address1              : a20
    var address2              : a20
    var city                  : a20
    var state                 : a2
    var zip                   : a10
    var tel                   : a12
    window 7, 33
    display 1, 2, "     Name:"
    display 2, 2, " Address1:"
    display 3, 2, " Address2:"
    display 4, 2, "     City:"
    display 5, 2, "    State:"
    display 6, 2, "      Zip:"
    display 7, 2, "Telephone:"

    displayinput 1, 13, name, "Enter name"
    displayinput 2, 13, address1, "Enter address1"
    displayinput 3, 13, address2, "Enter address2"
    displayinput 4, 13, city, "Enter city"
    displayinput 5, 13, state, "Enter state"
    displayinput 6, 13, zip, "Enter zip"
    displayinput 7, 13, tel, "Enter telephone number"
    windoweditwithsave
    txmsg "new_member", name, address1, address2, city, state, zip, tel
    waitforrxmsg

endevent
```

See Also

DisplayInput, **DisplayMSInput**, and **Window** commands

WindowScrollDown

Description

This command is used to scroll all lines in the current window down one line.

Syntax

WindowScrollDown

Remarks

- A **Window** command must have been executed prior to this command.
- The first line in the window is cleared to all spaces.

See Also

WindowScrollUp command

WindowScrollUp

Description

This command is used to scroll all lines in the current window up one line.

Syntax

WindowScrollUp

Remarks

- A **Window** command must have been executed prior to this command.
- The last line in the window is cleared to all spaces.

Example

The following example opens a file and reads from it one line at a time, displaying each line in a window. After the window is filled, for each line read from the file, previous lines in the window are scrolled up one row and the new line is displayed on the bottom row of the window.

```
event inq : 1
    var fn : n3
    var fname : a40 = "/micros/simphony/etc/script.isl"
    var line : a78
    var max_row : n3 = 14
    var row : n3

    fopen fn, fname, read                 //Open the file
    if fn = 0                             // if file can't be opened
    call ferr( fname )                    // let the use know
    endif
    window max_row, 78                    //Draw the window
    while not feof( fn )                  //Loop until the end
                                          // of file is encountered
        freadln fn, line                  // read a line from
                                          // the file
        call display_line( line )         //Call the display
                                          // subroutine, pass it the
line
    endwhile
    waitforclear


    (continued)
```

```
endevent
sub display_line( ref line )
   var col : n3 = 1

   if row < max_row                    //If row < max_row, the
                                       // window is not filled
   row = row + 1                       // increment the row
                                       //Counter so the following
   else                                // display command will
                                       // move to next row
      windowscrollup                   // otherwise, window's
                                       // full, scroll up
   endif
   display row, col, line              //Display the line
endsub


sub ferr( ref fname)
   exitwitherror "Can't open file ", fname
endsub
```

See Also

WindowScrollDown command

Chapter 8

ISL Functions

In This Chapter

This chapter summarizes all ISL Functions in and A-to-Z reference.

Functions

In addition to commands, ISL provides a variety of functions to enhance text handling and formatting facilities. Each function returns a value which may be useful for certain applications.

All function arguments must be enclosed in parentheses.

Function Summary

Below is a list of function names, a brief description, and the function type of the returned integer/character/key.

Function	Description	Function Type
Abs( )	Returns the absolute value of the integer or decimal	Integer
ArraySize( )	Returns the declared size of the array	Integer
Asc( )	Returns the ASCII integer value of the first character of the string	Integer
Bit( )	Returns the value of a bit in a hexadecimal string	Integer
Chr( )	Returns a one-character string that is the character representation of an integer	Character
Env( )	Returns a value of a shell environment variable	Integer
Feof( )	Determines if the file pointer is at the end of the file	Integer
FTell( )	Gets the file position in a file	Integer
GetHex( )	Converts a hex string to an integer	Integer
Instr( )	Returns the index of the first occurrence of a character string	Character
Key( )	Returns the key variable associated with the key pair	Key
KeyNumber( )	Returns the number portion of the key variable	Integer
KeyType( )	Returns the type portion of the key variable	Integer
Len( )	Returns the length of a string	Integer
Mid( )	Extracts part of a string to a string field	Character
ToInteger( )	Returns an integer from a decimal value	Integer
Trim( )	Returns a string trimmed of leading and trailing spaces	Character
VarSize( )	Returns the declared size of a variable	Integer

ISL Function Reference

This section includes all functions supported by the ISL in an A-Z reference format, which includes the following information for each function:

- **Description**: summarizes the function's purpose.
- **Syntax**: provides the proper way to specify the function and any arguments, as well as a description of each argument.
- **Remarks**: gives more detailed information of the function, its arguments, and how the function is used.
- **POS Setup**: provides any Simphony database programming required to issue the function successfully.
- **Example**: includes an example of the function being used in a script.
- **See Also**: names related functions, commands, system variables, and other documentation worth consulting.

Abs Function

Description

This function returns the absolute value of the integer or decimal value.

Syntax

Abs (*integer* or *decimal*)

Argument	Description
integer	an integer expression to be converted
decimal	a decimal expression to be converted

Example

```
event inq : 1

   var int : N5 = -145
   var mon : $8 = -12.35
   waitforclear abs(int)          //will display '145'
   waitforclear abs(mon)          //will display '12.35'
   waitforclear abs("-34")        //will display error
endevent
```

ArraySize Function

Description

This function returns the size (number of elements) of the array passed in.

Syntax

ArraySize (*array_name***)**

Argument	Description
array_name	the name of the array

Remarks

The array name must be placed between the parentheses without brackets. For example, the following references are illegal:

```
arraysize (list [ ])
arraysize (list [2])
```

The following entry is correct:

```
arraysize(list)
```

Example

The following subroutine returns the declared size (number of elements) of an array.

```
sub array_size
   var array_test[100] : a50
   waitforclear "Size of array_test is ", arraysize(array_test)
                                                    //Would prompt "Size
of
                                                    // array_test is
100"
endsub
```

Asc Function

Description

This function returns the ASCII integer value of the first character of the string passed in.

Syntax

Asc (*string_expression***)**

Argument	Description
string_expression	a place holder for text characters such as a *user_variable (string)*

Remarks

The process of returning a value with the **Asc** function works opposite of the **Chr** function.

Example

The following subroutine displays the ASCII value of the first character of a string:

```
sub asc_value
    var asc_val      : n3
    asc_val = asc("MICROS")
    waitforclear "The ascii value of M = ", asc_val
                                          //Would prompt "The ascii
                                          // value of M = 77"
endevent
```

See Also

Chr function

Bit Function

Description

This function will return the value (0 or 1) of a bit in a hexadecimal string.

Syntax

Bit (*hex_string, bit_position***)**

Argument	Description
hex_string	a place holder for the hexadecimal string that will be examined
bit_position	the index of the bit whose value is desired

Remarks

- The **Bit** function will generate an error if the string passed in contains non-hex characters. For example, the following statement will generate an error since the = character is not a hex digit:

```
i = gethex( "12AB=" )
```

- Valid hex digits are 0-9, A-F, and a-f.
- The *bit_positions* are numbered consecutively from 1. In the example below, a four-digit hexadecimal number would have bits numbered from 1 to 16. The digit values are determined by the standard hexadecimal assignment of bit values, i.e., within each digit, bit 1 = 8, bit 2 = 4, bit 3 = 2, bit 4 = 1.

Digit Position	**1**				**2**				**3**				**4**			
bit_position	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
digit value	**1**				**2**				**F**				**E**			
bit value	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**

Example

In the above example, the following command would result in `i` being set to 1.

```
i = bit("12FE", 7)                    //i will be set to 1
```

Thus in all hex_strings, *bit_position* 1 corresponds to the highest bit value in the first of the string.

Chr Function

Description

This function returns a one-character string that is the character representation of the integer passed in.

Syntax

Chr (*integer***)**

Argument	Description
integer	an integer in the range from 32 to 255

Remarks

The process of returning a value with the **Chr** function is the opposite of the **Abs** function.

Example

The following subroutine constructs the name of a POS company using ASCII values:

```
sub make_name
   var ascii_array[6]     : n3
   var pos_king           : a6
   var arr_cnt            : n3

   ascii_array[1] = 77
   ascii_array[2] = 73
   ascii_array[3] = 67
   ascii_array[4] = 82
   ascii_array[5] = 79
   ascii_array[6] = 83

   for arr_cnt = 1 to 6                //Count through the array
      format pos_king as pos_king, chr(ascii_array[arr_cnt])
   endfor

   waitforclear "The POS king is ", pos_king
endsub
```

See Also

Display command

Env Function

Description

This function returns the value of a shell environment variable.

Syntax

Env (*environment_variable***)**

Argument	Description
environment_variable	the environment variable to return

Remarks

An empty string will be returned if the environment variable does not exist.

Example

Assume that the environment variable “Term” is “ansi”:

```
var term : a20 = env("Term")      //term will be "ansi"
```

Feof Function

Description

This function tests whether the file pointer is at the end of the file.

Syntax

Feof (*file_number*)

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened

Remarks

A 1 is returned if there is no more data left to read, and a 0 is returned if there is more data left to be read.

Example

The following example shows how to use the **Feof** function as the condition of a **While** command:

```
while not feof( fn )
   call process_data( fn )
       .
       .
       .
endwhile
```

See Also

FClose, **FOpen**, and **While** commands

FTell Function

Description

This function returns the file position in a file.

Syntax

FTell (*file_number*)

Argument	Description
file_number	an integer variable which was assigned in the **FOpen** statement when the file was opened

Example

The following example will read a certain field position:

```
sub find_emp

   while not feof( fn )
   current_position = ftell( fn )       //Remember this position.

   fread fn, emp_number, *              //Read first field.
   if emp_number = target_emp_number    //If match, reposition back
      fseek fn, current_position        // to original position and
      return                            // let calling function
      endif                             // reread data.
   endwhile
endsub
```

See Also

FClose, **FOpen**, **FRead**, **FSeek**, and **While** commands

GetHex Function

Description

This function will convert a Hex string to a decimal integer.

Syntax

GetHex (*hex_string***)**

Argument	Description
hex_string	the *string* to be converted

Remarks

- The **GetHex** function will generate an error if the string passed in contains non-hex characters. The following statement will generate an error since the = character is not a hex digit, for example:

```
i = gethex( "12AB=" )
```

- A string should be made up of any combination of the following characters 0-9, a - f, or A - F.

Example

The following subroutine converts a hex string to its decimal equivalent:

```
sub hex_2_dec
   var hex_str     : a4 = "FFFF"
   waitforclear "The decimal equiv of 'FFFF' is ", gethex(hex_str)
endsub
```

Instr Function

Description

This function will return the index of the first occurrence of a character in a string.

Syntax

Instr (*index, string_expression, character***)**

Argument	Description
index	starting position of the search for the specified character
string_expression	the *string_expression* to be searched; can be one of the following: *user_variable* *system_variable* *constant* *string* *function* *equation*
character	character to search for

Example

The following statement will set `i` equal to 5, since "E" is the 5th character in the string:

```
i = instr( 2, "ABCDEFGHIJ", "E" )       //i will be set to 5
```

Key Function

Description

This function will execute the key function code defined.

Syntax

Key (*key_pair***)**

Argument	Description
key_pair	a link to a specific workstation key in the form of: *key_type : key_number*

Remarks

- The *key_type* determines the type of key (e.g., Function, Keypad); the *key_number* designates the specific Key Code.
- For a list of key codes and names, see “Key Types, Codes, and Names” on page D-1.

Example

The following script begins a check by number, then orders several menu items and prints the check:

```
event inq : 1

    loadkybdmacrokey (11,400), \                          //Begin check
               key (1, 552), key(1,554), key (1,555),\    //Order Menu
Items
               key (7, 101)                               //Service Total
                                                          // check

endevent
```

See Also

KeyNumber and **KeyType** functions

KeyNumber Function

Description

This function will return the key number (integer) portion of a key expression.

Syntax

KeyNumber (*key_expression***)**

Argument	Description
key_expression	an *expression* that can be one of the following: **@KEY** *...system_variable* *user_variable* **(KeyType)** **Key** function

Example

The following script reports the number and type of the [Enter] key:

```
event inq : 1
    var key_var  : a20 = "9,12"

    window 3, 26
    display 1, 2, "Enter key's key pair is", key_var
    display 1, 2, "Enter key's key type is", keytype(key_var)
    display 1, 2, "Enter key's key number is", keynumber(key_var) //keynumber = 12
    waitforclear
endevent
```

See Also

Key and **KeyType** functions

KeyType Function

Description

This function will return the key type (integer) portion of a key expression.

Syntax

KeyType (*key_expression***)**

Argument	Description
key_expression	an *expression* that can be one of the following: **@KEY** *...system_variable* *user_variable* **(KeyType)** **Key** function

Example

The following script reports the number and type of the [Enter] key:

```
event inq : 1
    var key_var  : a20 = "9,12"

    window 3, 26
    display 1, 2, "Enter key's key pair is", key_var
    display 1, 2, "Enter key's key type is", keytype(key_var)//key_type = 9
    display 1, 2, "Enter key's key number is", keynumber(key_var)
    waitforclear
endevent
```

See Also

Key and **KeyNumber** functions

Len Function

Description

This function is used to determine the length of a string or string variable.

Syntax

Len (*string_expression***)**

Argument	Description
string_expression	the string length to be returned

Example

The following script takes a list of names from the PMS and tests each name for its length. It then builds a window and displays the names. The longest string determines the width of the window.

```
event rxmsg : guest_list
    var guest_list_size : n3
    var guest_list[14]  : a78
    var arrcnt          : n3
    var longestr        : n3

    rxmsg guest_list_size, guest_list[]            //Receive size and list from PMS

    for arrcnt = 1 to guest_list_size              //Count through the array to
        if longestr < len(guest_list[arrcnt])      // find the longest member.
            longestr = len(guest_list[arrcnt])     //If this member is longer than
        endif                                      // the longest so far, set
    endfor                                         // longestr equal to its length.

    window guest_list_size, longestr + 2           //Build the window as high as
    for arrcnt = 1 to guest_list_size              //  the number of guests in the
        display arrcnt, 2, guest_list[arrcnt]      // list and as wide as the
    endfor                                         // longest guest name + 2.
    waitforclear                                   //Display the names and wait
endevent                                           // for user to press clear.
```

Mid Function

Description

This function is used to extract text from a string. This function is not to be confused with the **Mid** command.

Syntax

Mid (*string_expression*, *start, count***)**

Argument	Description
string_expression	a *string* or *user_variable (string)*
start	the starting offset (character) within the field
count	the number of characters to be read

Remarks

This command is similar to the BASIC language “mid$” function.

Example

The following subroutine searches a string for a specified character, starting at a specified place in the string, then returns the location of the character. The extracted text is compared to the desired characters. This example assumes that the calling routine declares the following four variables, and defines the first three:

```
//start:          n3              position in string to start search, 1 if not defined
// string        :a??             string to search
// search_char   :a1              character to search for
// charpos       :n3              the subroutine will set this variable equal to the
//                                location where the search_char is found in the
//                                string; it will be 0 if search_char is not found

sub instr
    if start <= 0                          //If user didn't define start
                                           // set it = 1
   start = 1
endif

for charpos = start to len(string)
   if search_char = mid(string, charpos, 1)                //If we find the
search_char,
                                                                  // return the
charpos
            return
        endif
    endfor
    charpos = 0                                    //If not found, set charpos = 0
endsub
```

See Also

Mid command

ToInteger Function

Description

This function returns an integer from a decimal value by removing the decimal point. This assumes that the new value will be interpreted correctly; i.e., the PMS will know where the decimal is placed.

Syntax

ToInteger (*decimal***)**

Argument	Description
decimal	a *decimal* expression to be converted

Remarks

The decimal point will be removed when returned.

Example

```
event inq : 1
   var n : N5
   n = 12.45                              //n will equal '12'
   n = tointeger( 12.45 )                 //n will equal '1245'
endevent
```

Trim Function

Description

This function is used to remove leading and trailing spaces from text or variable fields.

Syntax

Trim (*string_expression***)**

Argument	Description
string_expression	a *string* or *user_variable (string)*

Example

The following subroutine trims leading and trailing spaces from a string:

```
sub trim_spaces
   var string      : a32 = "   many spaces   "
   var trimmed_str : a11
   trimmed_str = trim(string)    //Result would be  "many spaces"
endsub
```

VarSize Function

Description

This function returns the declared size of a variable.

Syntax

VarSize (*user_variable***)**

Argument	Description
user_variable	the name of the *user_variable* to be "sized"

Example

The following script displays the declared size of a very large string:

```
event inq  : 12
   var big_string  : a200
   waitforclear "BIG_STRING'S size is ", varsize(big_string)
//would prompt "BIG_STRING'S size is 200"
endevent
```

Appendix A

ISL Error Messages

In This Chapter

This chapter explains the error messages returned by the ISL.

Error Message Format

Error messages will appear in the center of the workstation, in one of the formats described below. An explanation of the variable information referenced in the format syntax follows.

Variable Descriptions

<*error text*>: specifies a detailed explanation of what the error condition or syntax error may be.

<*line*>: specifies the line number where the error occurred.

<*column*>: specifies the column number where the error occurred.

<*error text* **( )**>: specifies a detailed explanation, including the specific erroneous data, enclosed in parentheses.

Format 1

ISL error
<error text>

Example

ISL error
No match for event

Format 2

ISL error on line *<line>*
<error text>

Example

ISL error on line 102
Command outside procedure

Format 3

ISL error on line *<line>*:*<column>*
<error text>

Example

ISL error on line 174: 12
Expected end of line (i)

Format 4

Another type of error occurs when the ISL expected specific text, but encountered different text. This error is displayed as:

ISL error on line <*line*>
expected <*text*>, encountered <*text*>

For example, if the **Format** command is issued, and the **as** (which is required as part of the syntax) is missing, the following error would display:

ISL error on line 170
expected 'AS' , encountered ' date'

Format 5

Format 5 is for the KWS (Keyboard Workstation) only.

ISL error:<*line number*>
<*error text*>

Example

ISL error :102
Command outside procedure

Error Messages

Array Index Out Of Range

This message occurs if an index number used to access a *list_array* is invalid.

```
var array[10] : N5
array[12] = 1                    //Valid range is 1-10
array[-2] = 4                    //No negative numbers
```

Bad sys var index

This message occurs if a *system_variable* size was out of range.

```
txmsg "MSG", @SI34               //Valid @SI range is 1-16
```

Break with too many endfor

This message occurs if too many **EndFor** commands occurred within a script without a corresponding **For** command.

```
for i = 1 to 10
   a[i] = 0
endfor
endfor                           //No corresponding
                                 //for command
```

Break without endfor

This message occurs if a **Break** command is issued within a **For** loop, but there is no **EndFor** command to complete the loop.

Call has no arguments

This message occurs if the subroutine called by the **Call** command has arguments, but no arguments were specified in the **Call** command.

```
event inq:1
   call mysub                    //Mysub has no arguments
endevent

sub mysub( var i:N5 )
      .
      .
      .
endsub
```

Can not evaluate

This message occurs if invalid text was encountered when trying to read an *expression*.

```
display 3, *, "Test"
```

Cannot access ISL script file

This message occurs if the script file was not found, or its permissions were not set correctly.

Command outside procedure

This message occurs if certain commands are issued outside an **Event** procedure.

```
window 10,20                    //This must be within the
                                // event procedure.
event inq:1
      .
      .
      .
endevent
```

The following commands are allowed outside an **Event** procedure:

```
ContinueOnCancel
DiscardGlobalVar
ExitOnCancel
Prorate
RetainGlobalVar
SetSignOnLeft
SetSignOnRight
UseBackUpTender
UseCompatFormat
UseISLFormat
UseISLTimeOuts
UseSTDTimeOuts
Var
```

Decimal overflow

This message occurs if an attempt to assign a value to a real exceeded the real's storage size.

```
var n : $3
n = 123.45                      //n only holds 3 digits
```

Display column or row out of range

This message occurs if the *row* and/or *column* declared with the **Display** command is outside the range declared by the **Window** command.

```
window 10, 20
display 11, 1, "Line"            //Row range is 1-10
display 1, 40, "Line"            //Column range is 1-20
```

Divide by zero

This message occurs if an attempt was made to divide a numeric value by 0.

```
i = a / 0
```

Duplicate variable def

This message occurs if an attempt is made to declare the same *variable* either within or outside an **Event** procedure.

```
event inq : 1
   var i : N5                    //First declaration OK.
   var i : A20                   //Redeclaration error.
      .
      .
      .
endevent
```

Encountered non-hex data

This message occurs if Hexadecimal data was expected and the *string* contained non-hex data.

```
i = gethex( "145B*" )            //* is not a hex character
```

Endsub nesting mismatch

This message occurs if an **Endsub** occurred within an **Event** command without its corresponding **Sub** command.

```
event inq : 1
      .
      .
      .
   endsub
endevent
```

Evaluation nesting

This message occurs if an overly complex *expression* was specified on the command line.

```
i = ((((((((((((((((( a + 5 ))))))))))))))))
```

Event inside procedure

This message occurs if an **Event** command was encountered within a subroutine.

```
sub check_message
   event inq : 1                    //Event within subroutine
      .
      .
      .
endsub
```

Event type must be word

This message occurs if an invalid **Event** type was specified in the **Event** line. There are four types of **Events**: Inquire, Tmed, RxMsg, and Final_Tender.

```
   event 123 : 377                  //No such event type as
                                    // 123
```

Exceeded max array or variable size

This message occurs if the *array_size* or *variable_size* exceeded the system maximum size of 32768 bytes.

```
var array[100000] : N5
```

Expected array in call

This message occurs if the **Sub** command had an *array* argument, but the **Call** command tried to pass a normal *variable*.

```
event inq:1
   var i:N5
   call mysub( i )                  //Has a normal variable
endevent

sub mysub( ref arr[] )              //Has an array
```

expected ..., encountered...

This message occurs when ISL receives unexpected text as part of the command syntax.

```
format date @DAY, "-" , month_arr[@MONTH], "-", @YEAR
                                    //"As" is missing after
                                    // the format command
```

Expected decimal

There are places in ISL where the script writer must specify a decimal number (and not an integer or a string). Using any expression other than a decimal expression results in this error.

For example: the function tointeger() expects a decimal number as its argument.

```
i:N5
i = tointeger( 12.34 )          //OK
i = tointeger( 1234 )           //Not ok. 1234 is not
                                // decimal.
i = tointeger( "12.34" )        //Not ok. "12.34" is
                                // not decimal.
```

Expected end of line

This message occurs if extraneous data was found at the end of a command line.

```
startprint 12 i                 // Data after 12 is an
                                // error
```

Expected format token

This message occurs if a *variable* was specified with the **Display** command that did not have a comma after it.

```
display 1, 2, i 123
```

Expected operand

This message occurs if an invalid *expression* was encountered.

```
var i : n5
   i = 5 +                      //Invalid expression
```

Expected string

This message occurs if a command or function expected a *string* as one of its arguments, and a *non-string expression* was encountered.

```
var i : n5
   setstring i, "a"             //n : 5 is a non-string
                                // expression
```

File buffer overflow

This message occurs if an attempt was made to read or write a line to a file, which exceeded the current @FILE_BFRSIZE.

```
@FILE_BFRSIZE = 10
fwrite fn, "This string is longer than 10 bytes"
```

File is read only

This message occurs if an attempt was made to write to a file opened for read access only.

File is write only

This message occurs if an attempt was made to read from a file opened for write access only.

File name too long

This message occurs if the file name in the **FOpen** command is greater than 128 characters.

Format needs string

This message occurs if the **Format** command requires a *string_variable* as its first argument.

```
var line : A15
format as "This is a line"        //Missing the variable
                                  // after format command
```

Format too long

This message occurs if the allocated size of the variable to be formatted is smaller than the total length of the expressions to be included.

```
var line : A10
format line as "This line is greater than 10 characters"
```

Integer overflow

This message occurs if an attempt to assign a value to an integer exceeded the integer's storage size.

```
var n : N3
n = 12345                    // n only holds 3 digits.
```

Invalid decimal operation

This message occurs if the operation is not allowed on real numbers. Real numbers are amounts, currencies, and decimals.

```
var a : $5, b : $5, c : $5
a = b % c
```

Invalid file buffer size

This message occurs if an attempt was made to assign the *system_variable* @FILE_BFRSIZE to an illegal value. An illegal value would be a value less than or equal to 0.

```
@FILE_BFRSIZE = -20
```

Invalid file mode

This message occurs if an invalid mode was specified on the **FOpen** command.

```
fopen fn, "test.log", read and wirte
                                    //The write mode is
                                    // misspelled
```

Invalid file number

This message occurs if a file number was passed to a File I/O command which was not previously opened.

```
event inq:1
   var fn:N5 = -4
                                    //No fopen declared
   fwrite fn, "hello"
endevent
```

Invalid first token

This message occurs if the start of a line contained invalid text.

```
display 2, 3, "Line"
*waitclear                          //* is invalid
```

Invalid input fmt spec

This message occurs if the input format specification contained invalid data.

```
input name{;}
```

Invalid list size

This message occurs if a command which required a list value encountered a list value of 0 or below.

Invalid locking mode

This message occurs if an invalid locking mode was specified in the **FLock** command.

```
flock fn, preventread and write   //Should be preventwrite
                                  // not write
```

Invalid output format

This message occurs if the output format specification contained invalid data.

```
txmsg name{;}
```

Invalid pms send

This message occurs if the system was unable to send the PMS message.

Length invalid

This message occurs if the third argument in the **Mid** command and/or function was less than 0.

List value too big

This message occurs if the list value in the command exceeded the *array* which it referenced.

```
var list[10] : A20
txmsg 21, list[]          // Valid range is 1-10
```

Loop variable constant

This message occurs if the **For** loop *variable* was not a *variable*.

```
for 10 = 1 to 20
```

Loop variable not int

This message occurs if the **For** loop *variable* was not an *integer*.

```
var i : $10
for i = 1 to 10
```

Max files open

This message occurs if an attempt was made to open more than 10 files in a single **Event**.

Max include nesting

This error occurs when include files become nested too deeply. Include files become nested when include files include other include files. For example, script.isl may include file1.isl, which may include file2.isl, and so forth. There is a limit to the depth of files that may be included.

This error can also occur when a file tries to include itself. In this case, the ISL interpreter keeps rereading the file at the point of inclusion, and continues until the file is read in 10 times. At this point, an error will be generated before the script has run.

Max lines executed

This message occurs if the *system_variable* @MAX_LINES_TO_EXECUTE was set to a non-zero value and @MAX_LINES_TO_EXECUTE command lines were executed.

Max macro keys

This message occurs if the maximum number of defined macro keys was encountered.

```
for i = 1 to 1000
   loadkybdmacro 1 = i
endfor
```

Max ref info

This message occurs if the maximum number of reference lines were issued by the **SaveRefInfo** command. The maximum number of lines is 8.

Max window input entries

This message occurs if too many **DisplayInput** entries were specified. The maximum number of **DisplayInput** entries is 64.

Memory allocation

This message occurs if an internal memory error has occurred.

Must have list var

This message occurs if ISL encountered a list specification without a list value.

```
txmsg list[]
```

Name is a reserved word

This message occurs if an attempt was made to declare a *variable* with the same name as a reserved word.

```
var display : n4
```

New tndttl exceeds original

This message occurs if an attempt was made to increase the *system_variable* @TNDTTL value, but the value can only be decreased.

No arrays in sub var

This message occurs if a **Sub** command tried to declare an *array_variable* in the argument list.

```
sub mysub( var i[ 10 ] : N5 )
```

No isl file

This message occurs if the script was not found or did not exist.

No match for endfor

This message occurs if no corresponding **EndFor** command exists for a **For** command.

```
event inq : 1
   for row_cnt = 1 to number_occupants
      display row_cnt, 2, occupant_list[row_cnt]
      .
      .
      .
                                        //Missing endfor command
endevent
```

No match for endwhile

This message occurs if no corresponding **EndWhile** command exists for a **While** command.

```
event inq:1
   .
   .
   .
   while i < 10

   .
   .
   .
                                        //No endwhile declared
endevent
```

No match for event

This message occurs if the SIM Inquiry and/or SIM Tender key had no corresponding Event Inq or Event_Tmed.

For example, if the SIM Inquiry Key #920 (SIM Key 1 : Inq 1) is pressed, and Event Inq : 1 does not exist, this message will display.

No number in sys var

This message occurs if a *system_variable* requires a number entry after it and no number was entered.

For example, *system_variable* @SI must have a number 1 - 16 after it.

No ops on strings

This message occurs if the declared *string* operation is not allowed.

```
a = "123" - "abc"
```

No pms message received

This message occurs if no response was received from the PMS system after the **RxMsg** command was executed.

No touchscreen keys defined

If the script executes a **ClearIslTs** command to clear the ISL-defined touchscreen, then immediately tries to display the ISL-defined touchscreen using **DisplayIslTs** or **PopUpIslTs**, the ISL cannot display the touchscreen because there are no keys to display.

```
event inq:1
   clearislts                    //Remove any  defined keys.
   displayislts                  //No keys, error occurs
                                 // here.
endevent
```

Not a variable

This message occurs if a *variable* was expected but not encountered.

```
input 123, "Enter value"         //123 not variable
```

Not enough input data

This message occurs if the @STRICT_ARGS *variable* is set and there were too many *variables* specified in the **RxMsg**, **Split**, **SplitQ**, or **FRead** command.

Assume the following data was received in a PMS message: Dan|Tooher. The message has two fields. The following example expects three fields and would generate the above error:

```
var fname : A20, lname : A20, status : N3
   .
   .
   .
rxmsg fname, lname, status
```

Not enough list data

This message occurs if the @STRICT_ARGS *variable* is set to a non-zero value, but the input data did not have enough values to assign the specified list.

Assume the following data was received in a PMS message: 3|Smith|Jones. The 3 signifies that three fields follow, and only two fields are present. The following would generate the above error:

```
var size : N3
var list[10] : A20
   .
   .
   .
rxmsg size, list[ ]
```

Not enough variables

This message occurs if the @STRICT_ARGS variable is set to a non-zero value, but there were not enough variables specified in the **RxMsg**, **Split**, **SplitQ**, or **FRead** command.

Assume the following data was received in a PMS message: DanTooher. The message has two fields. The following example expects one field and would generate the above error:

```
var fname : A20, lname : A20, status : N3
   .
   .
   .
rxmsg fname
```

Numeric entry required

This message occurs if non-numeric data was entered for a numeric variable.

Print already started

This message occurs if a **StartPrint** command was encountered while print was still active (i.e., prior to a corresponding **EndPrint**).

```
startprint @RCPT
   printline ...
startprint @CUST
      .
      .
      .
endprint
```

Print not started

This message occurs if a **PrintLine** or **EndPrint** command was encountered without a corresponding **StartPrint** command.

Reading ord dvc table

This message occurs if an error occurred while reading the *Order Devices* module in the Enterprise Management Console (EMC).

Reading tbl def

This message occurs if an error occurred while reading the *Tables* module in the EMC.

Require array for list

This message occurs if an *array* was expected but a *non-array* variable was encountered.

```
var i : N5
listdisplay 1, 2, 3, i            //i is not an array
```

Script memory allocation error

This message occurs if an internal error is encountered.

Start position invalid

This message occurs if the *start_position* parameter in the **Mid** command and/or function is invalid.

```
str = mid("abc", -2, 3)           //-2 is invalid
```

String overflow

This message occurs if an attempt to assign a value to a *string* exceeded the *string*'s storage size.

```
var n : A3
n = "message"                     //n only holds 3 characters
```

Sub array ref invalid

This message occurs if a **Sub** command had an invalid *array* declaration for an *array_ variable*.

```
sub mysub( ref array[ )           //Only one bracket
                                  //Should be [ ]
```

Sub has no arguments

This message occurs if a **Call** command was made with arguments, and the subroutine called had no arguments.

```
event inq:1
   call mysub( 1, 2, 3 )
endevent

sub mysub                          //Mysub missing ( 1, 2, 3 )
   .
   .
   .
endsub
```

Sub statement in procedure

This message occurs if a **Sub** command was encountered while inside an **Event**.

```
event inq:1
   sub mysub
      .
      .
      .
   endsub
endevent
```

System variable declaration

This message occurs if an attempt was made to declare a *system_variable* (i.e., any variable name that begins with the @ character).

```
var @chk : N3
```

Sys var not assignable

This message occurs if an attempt was made to assign a value to a read-only *system_variable*.

Too few args in call

This message occurs if the **Call** command did not have enough arguments.

```
event inq:1
   call mysub( 1 )
endevent

sub mysub( var i:n5, var j:N5 )
      .
      .
      .
endsub
```

Too few arguments

This message occurs if there were not enough arguments specified for a function.

Too many args in call

This message occurs if the **Call** command had too many arguments.

```
event inq:1
   call mysub( 1, 2 )
endevent

sub mysub( var i:n5 )
      .
      .
      .
endsub
```

Too many arguments

This message occurs if too many arguments were specified for a function.

Too many nested calls

This message occurs if too many subroutines were nested within each other.

Too many touchscreen keys

The ISL-defined touchscreen will hold a finite number of keys (i.e, nine). If the user tries to define too many keys, this error will occur.

```
event inq:1
   var i:n5

   clearislts              // Remove any defined keys.
   for i = 1 to 100        // Loop will generate an error.
      setisltskey 1, 1, 2, 1, 1, @key_clear, "CLEAR"
   endfor
   displayislts            // No keys, error occurs here.
endevent
```

Too many PMS definitions active. Start new transaction

This message occurs if the following condition occurs: the Revenue Center PMS link database file must have changed while the User Workstation was in a transaction. To clear this condition, cancel the current transaction.

Undefined call

This message occurs if a **Call** was made to a subroutine that did not exist within the script.

Undefined function

This message occurs if an undefined function was called.

Undisplayable variable

This message occurs if the *variable* cannot be displayed.

```
display 2, 3, @TRDTL              //@TRDTL can not be
                                  // displayed
```

Unexpected data after call

This message occurs if the **Call** command is invalid.

```
call mysub + 3
```

Unexpected data after sub

This message occurs if the **Sub** command is invalid.

```
sub mysub - 4
```

Unexpected data in sub

This message occurs if the parameter list in the **Sub** command is invalid.

```
sub mysub( var fred:N5, i : N5 )
```

Unexpected end of line

This message occurs if not enough data was specified on the command line.

```
display 2, 2,             //should be data after 2,
```

Unexpected token type

This message occurs if invalid text is encountered when trying to read a command or function.

Unknown command

This message occurs if an unknown command is specified.

```
dsplay 2, 2, "Line"       //display is misspelled
```

Unknown system variable

This message occurs if an unknown *system_variable* is referenced.

```
display 2, 2, @RVVC
```

Unmatched endevent

This message occurs if an **Endevent** was encountered without a corresponding **Event** command.

Unmatched endfor

This message occurs if a **For**/**EndFor** nesting error occurred.

Unmatched if

This message occurs if an **If**, **ElseIf**, **Else**, or **EndIf** nesting error occurred.

Value not key definition

This message occurs if an attempt to use a non-key *variable* in an *expression* which required a *key_variable* was encountered.

```
loadkybdmacro 12.47
```

Variable undefined

This message occurs if an undefined *variable* was referenced.

Window columns out of range

This message occurs if an attempt was made to declare a **Window** that was too wide.

```
window 4, 1000
```

Window has not been defined

This message occurs if an attempt to display text within a **Window** occurred without a **Window** first being declared.

Window rows out of range

This message occurs if an attempt was made to declare a **Window** that was too tall.

```
window 1000, 10
```

Appendix B

TCP Interface Code

In This Chapter

This chapter includes sample code for MICROS SIM TCP Server, Sample SIM Server, and a sample makefile.

MICROS SIM TCP Server

```
/*
 * MICROS SIM TCP Server
 *
 * This code implements a server process which accepts SIM messages
 * from an Oracle MICROS POS client process over a TCP link.
 *
 * This sample code is written for UNIX System V using the AT&T SVID
 * Transport Layer Interface (TLI) API. It should be easily portable
 * to the X/Open Transport Interface (XTI). Porting to a
 * Berkeley-style socket library is left as an exercise for the
 * reader.
 *
 */

#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <sysexits.h>
#include <sys/types.h>

#include <netdb.h>
#include <tiuser.h>
#include <stropts.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/netinet/in.h>

extern int t_errno;

/* operating-system specific device name: */
#define TCP_DEVICE_NAME  "/dev/inet/tcp"

/* define SIM TCP service: */
#define SIM_SERVICE_NAME "micros-sim"
#define SIM_SERVICE_TYPE "tcp"
#define DEFAULT_SIM_PORT 5009

#define SIM_MAX_MSG            32767
#define SIM_MAX_MSG_BODY       (32767 - 25 - 4 - 4)

/* supplied by SIM vendor: */
extern void process_pos_request(const char *header,
                                const char *body,
                                char reply_body[SIM_MAX_MSG_BODY]);

/* supplied below: */
static void transfer_pos_messages(int fd);

void run_sim_server(void)
```

```
{
   int listen_fd, conn_fd;
   struct sockaddr_in *sin;
   struct servent *servp;
   struct t_bind *bind;
   struct t_call *call;
   u_short serviceport;
   int retries;

   /* Open a TCP server endpoint in order to listen for
    * requests from POS client processes.
    *
    * If the requested address is in use, retry up to 10 times.
    * This can occur if another server was running on the same
    * address, and the TCP port has not yet completed its shutdown
    * processing.
    */

   if ((servp = getservbyname(SIM_SERVICE_NAME, SIM_SERVICE_TYPE)) ==
NULL)
      serviceport = htons(DEFAULT_SIM_PORT);
   else
      serviceport = (u_short) servp->s_port;

   retries = 10;
   while (retries--) {

      if ((listen_fd = t_open(TCP_DEVICE_NAME, O_RDWR, NULL)) < 0) {
         t_error("run_sim_server: t_open");
         exit(EX_OSFILE);
      }

      if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL))
          == NULL) {
         t_error("run_sim_server: t_alloc(T_BIND)");
         t_close(listen_fd);
         exit(EX_OSERR);
      }

      sin = (struct sockaddr_in *) bind->addr.buf;
      sin->sin_family      = AF_INET;
      sin->sin_addr.s_addr = INADDR_ANY;
      sin->sin_port        = serviceport;
      bind->addr.len       = sizeof *sin;
      bind->qlen           = 1;

      if (t_bind(listen_fd, bind, bind) < 0) {
         t_error("run_sim_server: t_bind");
         t_close(listen_fd);
         exit(EX_OSERR);
      }
```

```
      if (sin->sin_port != serviceport) {
         fprintf(stderr,
                 "run_server: wanted port %d, got port %d, retrying\n",
                 ntohs(serviceport), ntohs(sin->sin_port));
         t_close(listen_fd);
         t_free((char *)bind, T_BIND);
         sleep(10);
      }
      else
         break;

   }

   if (retries == 0) {
      fprintf(stderr, "run_sim_server: could not get port %d\n",
              ntohs(serviceport));
      t_close(listen_fd);
      exit(EX_TEMPFAIL);
   }

   if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL)) ==
NULL) {
      t_error("run_sim_server: t_alloc(T_CALL)");
      t_close(listen_fd);
      exit(EX_OSERR);
   }

   /* For simplicity, we ignore SIGCLD, which allows the exiting
    * child processes to clean up after themselves, without
    * requiring the parent (this process) to call wait().
    */

   sigignore(SIGCLD);

   /* We now have the desired TCP port open.
    * Accept connections, and for each connection accepted,
    * start a server process.
    */

   while (1) {

      /* Listen for incoming connections.
       * This process will typically spend 99.9% of its time
       * blocked in this t_listen() call.
       */

      if (t_listen(listen_fd, call) < 0) {
         t_error("run_sim_server: t_listen");
         t_close(listen_fd);
         exit(EX_OSERR);
      }
```

```
      /* Open a new endpoint and accept the connection
       * on this new endpoint (freeing the listen_fd
       * to accept further connections).
       */

      if ((conn_fd = t_open(TCP_DEVICE_NAME, O_RDWR, NULL)) < 0) {
         t_error("run_sim_server: t_open");
         t_close(listen_fd);
         exit(EX_OSFILE);
      }

      if (t_bind(conn_fd, NULL, NULL) < 0) {
         t_error("run_sim_server: t_bind");
         t_close(conn_fd);
         t_close(listen_fd);
         exit(EX_OSERR);
      }

      if (t_accept(listen_fd, conn_fd, call) < 0) {
         if (t_errno == TLOOK) {
            /* retrieve disconnect indication, if any, and continue */
            t_rcvdis(listen_fd, NULL);
            t_close(conn_fd);
         }
         else {
            t_error("run_sim_server: t_accept");
            t_close(conn_fd);
            t_close(listen_fd);
            exit(EX_OSERR);
         }
      }
      else {

         /* Push the "tirdwr" module onto the connection, establishing
          * the "read/write" interface.  This is so the rest of this
process
          * does not have to understand the more complicated TLI scheme
          * for pushing messages, and can treat this connection just like
          * a tty.  (Let the streams module do the work.)
          *
          * After this succeeds, no TLI calls can be made on conn_fd,
          * only read(), write(), and close().
          */

         if(ioctl(conn_fd, I_PUSH, "tirdwr") < 0) {
            perror("run_sim_server: ioctl(I_PUSH, tirdwr)");
            t_close(conn_fd);
            t_close(listen_fd);
            exit(EX_OSERR);
         }

         /* Start a child process, which will use conn_fd.
```

```
          * The parent will close conn_fd and return to
          * listening.  If the fork fails, we will discard
          * this connection, but continue to listen,
          * since the situation should clear up eventually.
          */

         switch(fork()) {

          case -1:  /* error */
            perror("run_sim_server: fork");
            close(conn_fd);
            break;

          default:  /* parent */
            close(conn_fd);
            break;

          case 0:   /* child */
            t_close(listen_fd);
            transfer_pos_messages(conn_fd);
            exit(EX_OK);
            break;

         }
      }
   }
}

#define SOH   1
#define STX   2
#define ETX   3
#define EOT   4
#define ACK   6
#define NAK  21

enum SIM_Link_State { sim_msg_begin, sim_msg_id, sim_msg_data,
sim_msg_cksum };

static void transfer_pos_messages(int fd)
{
   int i, n;
   int msg_buf_len;
   char *header, *body;
   enum SIM_Link_State state;
   char msg_buf[SIM_MAX_MSG + 1];
   char recv_buf[SIM_MAX_MSG + 1];
   char reply_buf[SIM_MAX_MSG + 1];
   char reply_body[SIM_MAX_MSG_BODY];



   /* Handle input from POS system, implementing the network protocol:
```

```
     *   1. Attempt to read bytes from POS connection, blocking.
     *   2. If an invalid message is received, discard it.
     *   3. When a complete message is available, call
     *      process_pos_request,
     *      passing the received header, the received body,
     *      and a buffer for the reply.
     *   4. When process_pos_request returns, reply to the POS
     *      connection with a copy of the received header and
     *      the reply buffer returned from process_pos_request.
     */

   state = sim_msg_begin;
   msg_buf_len = 0;

#define NEXT_STATE(prev,next) (state == (prev) \
                               ? (state = (next), 1) \
                               : (state = sim_msg_begin, 0))

#define STATE_ERROR {  msg_buf_len = 0; state = sim_msg_begin; }

   while (1) {

      n = read(fd, recv_buf, SIM_MAX_MSG);

      if (n < 0 && errno == EINTR)
         continue;  /* ignore interrupts, read again */

      if (n < 0) {
         perror("transfer_pos_messages: read");
         close(fd); /* this connection is finished */
         return;
      }

      if (n == 0) {
         fprintf(stderr, "transfer_pos_messages: connection closed\n");
         close(fd); /* this connection is finished */
         return;
      }

      for(i = 0; i < n; i++) {

         switch(recv_buf[i]) {

          case SOH:
            if(NEXT_STATE(sim_msg_begin, sim_msg_id)) {
               /* Restart message, and, for clarity,
                * remember the position of the (upcoming) first
                * byte of the header.
                */
               msg_buf_len = 0;
               header = &msg_buf[msg_buf_len];
            }
```

```
  else
     STATE_ERROR
  break;

case STX:
  if(NEXT_STATE(sim_msg_id, sim_msg_data)) {
     /* NUL-terminate header, and remember the position
      * of the (upcoming) first byte of the body
      */
     msg_buf[msg_buf_len++] = '\0';
     body = &msg_buf[msg_buf_len];
  }
  else
     STATE_ERROR
  break;

case ETX:
  if (NEXT_STATE(sim_msg_data, sim_msg_cksum)) {
     /* NUL-terminate the body. */
     msg_buf[msg_buf_len++] = '\0';
  }
  else
     STATE_ERROR
  break;

case EOT:
  if(NEXT_STATE(sim_msg_cksum, sim_msg_begin)) {

     int reply_len;
     reply_body[0] = '\0';



     /* If body exists,
      *    send request to SIM-specific code.
      * header[0] through NUL is the header.
      * body[0] through NUL is the request body.
      * reply_body[0] through NUL will be the reply body.
      * (If body is empty, return empty response to POS.)
      */

     if (body[0] != '\0')
        process_pos_request(header, body, reply_body);

     /* Frame original header and reply body,
      * and respond to the POS system.
      */

     sprintf(reply_buf, "%c%s%c%s%c%c",
             SOH, header, STX, reply_body, ETX, EOT);
     reply_len = strlen(reply_buf);
     if (write(fd, reply_buf, reply_len) != reply_len) {
```

```
                perror("transfer_pos_messages: write");
                close(fd);
                return;
             }

          }
          else
             STATE_ERROR
          break;

        case ACK:
        case NAK:
          /* Ignore ACKs and NAKs */
          break;

        default:
          switch(state) {
           case sim_msg_begin:
           case sim_msg_cksum:
           default:
             break;
           case sim_msg_id:
           case sim_msg_data:
             msg_buf[msg_buf_len++] = recv_buf[i];
             break;
             break;
          }
          break;

      }
    }
  }
}
```

Sample SIM Server

```
/*
 *
 * This code is a complete implementation of a SIM server,
 * demonstrating the functions which must be provided by
 * the server application.
 *
 * This example sends valid responses to Simphony-standard format POS
messages.
 */

extern void run_sim_server(void);

main()
{
 run_sim_server();
}

void process_pos_request(const char *header,
                   const char *body,
                   char reply_body[])
{
   printf("Header = %s\n", header);
   printf("Body = %s\n", body);
   if (body[0] == ' ' && body[1] == '1')
      strcpy(reply_body, " 1ABCDEFGHIJKLMNOP");
   else if (body[0] == ' ' && body[1] == '2')
      strcpy(reply_body, " 2ZYXWVUTSRQPONMLK");
   else
      strcpy(reply_body, "///UNKNOWN REQUEST");

   printf("Reply Body = %s\n", reply_body);
}
```

Sample Makefile

```
Simtest: Simtest.o Simsrv.o
      cc -o simtest simtest.o simsrv.o -lsocket -lnsl_s -lc_s

simtest.o: simtest.c
      cc -c -W2 -strict -g simtest.c

simsrv.o: simsrv.c
      cc -c -W2 -strict -g simsrv.c
```

Appendix C

ISL Quick Reference

In This Chapter

This chapter is a quick reference guide to the syntax of all ISL language elements, including data types, operators, system variables, format specifiers, commands, and functions.

Data Types

Data Type	Abbreviation	Description	Example
Numeric	N*x*	These variables are used for numeric information and may comprise *x* digits (integers, not decimal), e.g., N4 supports -9999 to 9999.	var rowcnt : n3
Decimal	$*x*	These variables are used for decimal amounts. Operator entries will assume a decimal place according to the currency's default setting, as specified in the *Currency* module; i.e., entering 1234 in the US will result in an amount of 12.34. They may comprise *x* digits, e.g., $4 in the US will support -99.99 to 99.99.	var new_ttl : $12
Alphanumeric	Returns the ASCII integer value of the first character of the string	These variables may include any non-control character, including punctuation marks. They may comprise *x* characters.	var name : a20
Key	key	This system variable is used for key press variables.	var keypressed : key

Relational and Logical Operators

Unary Operators

Operator	Description	Example
-	Negation operator (a minus sign). This is used to negate an expression.	-3 -count -((count + 5) * -index)
NOT	Will negate the result of the expression. The NOT operator can be applied to expressions in the same way as the unary minus operator.	The NOT operator will negate the result of the expression. For example, the following expression is always TRUE: (3 < 4) The NOT operator will negate the sense of the above expression; thus the following expression is always FALSE: NOT (3 < 4)

Binary Operators

Operation	Operator	Allowable Operand Types: Nx, $x, and Ax
multiplication	*	**N***x*, **$***x* **A***x*, **Key**
division	/	**N***x*, **$***x*
modulus	%	**N***x*, **$***x*
plus	+	**N***x*, **$***x*
minus	-	**N***x*, **$***x*
bit-wise and	&	**N***x*
bit-wise or	|	**N***x*

Operation	Operator	Allowable Operand Types: Nx, $x, and Ax
equality	**=**	**N***x*, **$***x* **A***x*, **Key**
greater than or equal	**>=**	**N***x*, **$***x*, **A***x*, **Key**
greater than	**>**	**N***x*, **$***x*, **A***x*, **Key**
less than or equal	**<=**	**N***x*, **$***x*, **A***x*, **Key**
inequality	**<>**	**N***x*, **$***x*, **A***x*, **Key**
less than	**<**	**N***x*, **$***x*, **A***x*, **Key**
logical and	**AND**	**N***x*
logical or	**OR**	**N***x*

System Variables

Category	Variable Name and Syntax	Field/Parameter
Cover/Guest Count	@GST	Guest Count
Credit Card	@CCDATE	Credit Card Expiration Date
	@CCNUMBER	Credit Card Account Number
Data Entry	@FIELDSTATUS	Data Entry Field Status Flag
	@INPUTSTATUS	User Input Status Flag
	@MAGSTATUS	Magnetic Card Entry Status Flag
	@RETURNSTATUS	Transaction Item Return Indicator
	@USERENTRY	Data Entered Before SIM Inquiry Key Activated
	@VOIDSTATUS	Transaction Item Void Indicator
Date and Time	@DAY	Current Day of Month
	@EPOCH	EPOCH Time
	@HOUR	Current Hour of Day
	@MINUTE	Current Minute
	@MONTH	Current Month
	@SECOND	Current Second
	@WEEKDAY	Day of Week
	@YEAR	Current Year
	@YEARDAY	Current Day of Year

Category	Variable Name and Syntax	Field/Parameter
Discount/Service Charge	@AUTOSVC	Auto Service Charge
	@CHGTIP	Charged Tip
	@DSC	Discount Total
	@DSC_OVERRIDE	When a manual discount is entered, a SIM 'Discount' script can decrease the amount of the discount by setting this variable to the desired discount amount
	@DSCI	Discount Itemizer
	@EMPLDISCOUNT	In a discount event, this variable is the number of the employee discount
	@EMPLDISCOUNTEMPL	In a discount event, this variable is the employee number of the discount receiving the employee discount
	@NUMDSC	Active Discounts
	@NUMSVC	Active Service Charges
	@SVC	Service Charges
	@SVCI	Service Charge Itemizer
Employee	@CKEMP	Check Employee
	@EMPLOPT[]	SIM ISL Options #1-#8
	@TRAININGMODE	Training Mode Status Flag
	@TREMP	Transaction Employee
Event Data	@EVENTID	ID of event being raised
	@EVENTTYPE	Type of event being raised
	@OBJ	Object number of detail item
	@PICKUPLOAN	Value of pickup or loan amount
	@QTY	Quantity of detail item
	@TTL	Amount of detail item
File I/O	@FILE_BFRSIZE	User Definable Variable
	@FILE_ERRNO	Standard Error Number Value
	@FILE_ERRSTR	Standard Error String based on @FILE_ERRNO
	@FILE_SEPARATOR	Field Separator for File I/O Operations

Category	Variable Name and Syntax	Field/Parameter
Function Keys	@KEY_CANCEL	Cancel Key
	@KEY_CLEAR	Clear Key
	@KEY_DOWN_ARROW	Arrow Down Key
	@KEY_END	End Key
	@KEY_ENTER	Enter Key
	@KEY_EXIT	Exit Key
	@KEY_HOME	Home Key
	@KEY_LEFT_ARROW	Arrow Left Key
	@KEY_PAGE_DOWN	Page Down Key
	@KEY_PAGE_UP	Page Up Key
	@KEY_RIGHT_ARROW	Arrow Right Key
	@KEY_UP_ARROW	Arrow Up Key
Guest Check	@CHK_OPEN_TIME	Date and Time Check Opened
	@CHK_OPEN_TIME_T	Current Check Open Time
	@CHK_PAYMNT_TTL	Current Check Payment Total
	@CHK_TTL	Current Check Total
	@CHECKDATA	Facsimile of Check
	@CKCSHR	Cashier Number
	@CKID	Check ID
	@CKNUM	Check Number
	@LASTCKNUM	Last Check Number Assigned to Guest Check
	@TRCSHR	Transaction Cashier Number
Open Check	@NUM_OPENCHECKS	Lists Open Checks per Revenue Center
	@OPENCHECK_EMPOWNER	Open Check Employee Object Number
	@OPENCHECK_GUID	Open Check GUID
	@OPENCHECK_NUMBER	Open Check Number
	@OPENCHECK_OPENTIME	Open Check Date and Time that the check was begun
	@OPENCHECK_ORDERTYPE	Open Check Order Type ID
	@OPENCHECK_TOTAL	Open Check Total Amount

Category	Variable Name and Syntax	Field/Parameter
	@OPENCHECK_WSOWNER	Open Check Workstation ID
Order Type	@ORDERTYPE	Order Type
Printing	@CHK	Guest Check Printer
	@DWOFF	Double-wide Characters OFF
	@DWON	Double-wide Characters ON
	@HEADER	Guest Check, Receipts, Credit Card Vouchers
	@NUL	Specifies a binary 0 should be sent
	@ORDR[]	Remote Order or Local Order Printer
	@PRINTSTATUS	Print Status Flag
	@RCPT	Customer Receipt Printer
	@REDOFF	Red Ink OFF
	@REDON	Red Ink ON
	@TRAILER	Guest Check, Receipts, Credit Card Vouchers
	@VALD	Validation Chit Printer
Property Management System (PMS)	@OFFLINELINK	Used to link to an offline PMS system
	@PMSLINK	Revenue Center PMS Link
	@PMSNUMBER	PMS Object Number
	@RXMSG	Name of PMS Response Message
	@SIMDBLINK	Links to the SIMDB DLL to the database
Proration	@GSTRMNG	Guests Remaining after Proration
	@GSTTHISTENDER	Guest Count Associated with Split Tender
Sales Itemizer	@NUMSI	Active Sales Itemizers
	@SI[]	Sales Itemizers
	@TXBL[]	Taxable Sales Itemizers
Sales Total	@CHANGE	Change Due
	@PREVPAY	Previous Payment
	@TNDTTL	Tender Total
	@TTLDUE	Total Due

Category	Variable Name and Syntax	Field/Parameter
Script	@RANDOM	Returns a random value between 0 and 2^{32}-1
	@RVC	Revenue Center Number
	@STRICT_ARGS	Strict Arguments
	@VARUSED	Used Variable Space
	@WARNINGS_ARE_FATAL	Strong Checking
Seat	@SEAT	Active Seat Number
Serving Period	@SRVPRD	Serving Period
System	@DBVERSION	Current Database Version
	@GUID	The GUID of the Current Check
	@OS_PLATFORM	1 - Windows® CE 3 - Win 32
	@PLATFORM	Hardware Platform
	@PROPERTY	The Property Number of the Workstation
	@SYSTEM_STATUS	Shell Return Status
	@VERSION	SIM Version Number
Table	@GRPNUM	Table Group Number
	@TBLID	Table ID
	@TBLNUM	Table Object Number
Tax	@NUMTAX	Active Tax Rates
	@RVCSERIALNUM[]	Revenue Center Sequence Number
	@SYSSERIALNUM[]	System Sequence Number
	@TAX[]	Tax Collected
	@TAXRATE[]	Tax Rate
	@TAXVAT[ ]	Returns the Value Added Tax Amount for Tax Rate “X”
	@TXEX_ACTIVE[ ]	Checks if the Tax is Exempt at the Specified Level
Tender/Media	@SIGCAPDATA	Signature Capture Data
	@TMDNUM	Tender/Media Number
Touchscreen	@ALPHASCREEN	Alpha Touchscreen
	@NUMERICSCREEN	Numeric Touchscreen

Category	Variable Name and Syntax	Field/Parameter
Transaction Detail	@DETAILSORTED	Detail Sorting Status
	@DTL_CAACCTINFO[ ]	Credit Authorization Account Information
	@DTL_CABASETTL[ ]	Credit Authorization Base Total
	@DTL_CAEXPDATE[ ]	Credit Authorization Expiration Date
	@DTL_CATIPTTL[ ]	Credit Authorization Tip Total
	@DTL_CATMEDOBJNUM[ ]	Credit Authorization Tender/Media Object Number
	@DTL_DEFSEQ[]	Definition Sequence of Detail Item
	@DTL_DSC_EMPL[ ]	Employee who is getting the employee meal discount for the specified detail entry
	@DTL_DSCI[ ]	Menu Item Detail Class Discount Itemizer Value
	@DTL_FAMGRP[]	Menu Item's Family Group
	@DTL_INDEX	Index of the detail which fired the SIM event
	@DTL_IS_COND[i]	Determines if a Guest Check Menu Item is a condiment
	@DTL_MAJGRP[]	Menu Item's Major Group
	@DTL_MLVL[]	Main Menu Level of Detail Item
	@DTL_NAME[]	Name of Detail Item
	@DTL_OBJNUM[]	Object Number of Detail Item
	@DTL_PLVL[]	Price Level of Detail Item
	@DTL_PMSLINK[]	PMS Link of Detail Item
	@DTL_PRICESEQ[]	Price Sequence Number of Detail Item
	@DTL_QTY[]	Quantity of Detail Item
	@DTL_SEAT[]	Seat Number of Detail Item
	@DTL_SLSI[ ]	Menu Item Detail Class Sales Itemizer Value
	@DTL_SLVL[]	Sub-menu Level of Detail Item
	@DTL_STATUS[]	Status of Detail Item
	@DTL_SVC_LINK[ ]	Stored Value Card Link
	@DTL_SVC_TYPE[ ]	Stored Value Card Type

Category	Variable Name and Syntax	Field/Parameter
Transaction Detail continued	@DTL_TAXTTL [ ]	Returns the Total Tax Amount for the Detail
	@DTL_TAXTYPE[]	Tax Types
	@DTL_TTL[]	Total of Detail Item
	@DTL_TYPE[]	Type of Detail Item
	@DTL_TYPEDEF[ ]	Returns the Detail Item Type Definition
	@MAXDTLR	Maximum Size of @TRDTLR
	@MAXDTLT	Maximum Size of @TRDTLT
	@NUMDTLR	Number of Detail Entries this Service Round
	@NUMDTLT	Number of Detail Entries for Entire Transaction
Troubleshooting	@LINE	Current Line Executed in Script
	@LINE_EXECUTED	Lines Executed in Script
	@MAX_LINES_TO_RUN	Maximum Lines of Script to Execute
	@PMSBUFFER	PMS Message
	@SHOW_PMS_MESSAGES	PMS Status Flag
	@TRACE	Output Line of Script to 8700d.log
Window	@CENTER	Center Column in ISL-defined Window
	@WCOLS	Number of Columns in ISL-defined window
	@WROWS	Number of Rows in ISL-defined window
Workstation	@WSID	Workstation ID number
	@WSTYPE	User Workstation Type
	@WSSUBTYPE	Use Workstation Subtype

Format Specifiers

Input Specifiers

Input Specifier	Description
-	Data being typed in by the operator should not be echoed back to the display Example: Input auth_code**{-}**, "Enter authorization code"
M*n*,*	Specify the track number (*n* = 1 or 2) and what data to read from the magnetic card. For use with the **Input**, **InputKey**, **DisplayInput**, and **DisplayMSInput** commands only. The M character is case-insensitive Example: Input auth_code{ M2,* }, "Enter authorization code"
M*n*, *field*, *start*, *count* | *	M*n:* the track number (M1 or M2); this can be followed by a star (*) to specify all fields on the track, or use the following fields to read specific information: *field:* the field position within the specified track; this is a positive integer *start:* starting offset (character) within the field;for example, if one wants to take the lastfour characters of the "Blaine Richard" string, the offset would start at 11 *count:* number of characters to be read from the *start* (first character) to the end of the *field* (place an asterisk * to include all characters*)* Example: Input auth_code{ M2,1,3,10 }, \ "Enter authorization code"

Output Specifiers

The proper syntax for using the *output_specifiers* is as follows:

[<|=|>|*] [+] [0] [*size*] [D|X|O|B] [^] ["] [:*format_string*]

Output specifiers ***must*** also be placed in the order listed in the following table:

Output Specifier	Description
<	Left justification; the *size* specifier may be used to specify the size of the field.
=	Center justification; the *size* specifier may be used to specify the size of the field.
>	Right justification; the *size* specifier may be used to specify the size of the field.
*	Trim leading and trailing spaces; the *size* specifier may be used to specify the size of the field.
+	Place sign at the start of the field.
0	Pad with zeroes (as opposed to spaces).
size	Where *size* is the number of the characters in the required field. The *size* must be a positive integer or an expression that is a positive integer.
D	Decimal (Default); display numerics in decimal format.
X	Hexadecimal; display numerics in hexadecimal format.
O	Octal; display numerics in octal format.
B	Binary; display numerics in binary format.
^	Place a space on each side of the data to be displayed.
"	Place double quotes around the data to be displayed.
:*format_string*	Similar to the BASIC language PRINT USING command. All characters will be displayed except for the # character, which will be replaced by characters from the variable or expression preceding the format specifier.

Commands

Note: *All arguments enclosed in brackets [ ] are considered optional.*

Category	Command and Syntax	Description
Communications	**QueueMsg** pms_number, expression[{output_specifier}][, expression[{output_specifier}]...]	Hold PMS messages in the Simphony database queue until the PMS is online.
	ReTxMsg	Retransmit a message.
	RxMsg user_variable or list_spec[, user_variable or list_spec...	Define the format of a message received over the interface.
	TxMsg expression[{output_specifier}][, expression[{output_specifier}]...]	Define the format and send an interface message.
	TxMsgOnly *expression*[{*output_specifier*}][, *expression*[{*output_specifier*}] \...]	Send a message to a PMS without waiting for a response.
	Use[Compat/ISL]Format	Use Simphony-standard or ISL message format.
	Use[ISL/STD]TimeOuts **UseTMSFormat**	Use ISL time outs or the standard Simphony error messaging, when there is no response from the PMS System. Format messages using the TMS format.
	WaitForRxMsg [prompt_expression[{output_specifier}]\ [, prompt_expression[{output_specifier}]...]	Wait for an interface message to be received after a TxMsg has been sent. If no prompt text is supplied, `Please Wait--Sending Message` is the default.

Category	Command and Syntax	Description
File I/O	**FClose** file_number	Close a file.
	FGetFile file_number	Gets a file from the SIM file service.
	Flock file_number, **Preventwrite** [**And**] [**Preventread**] [**and**] [**Nonblock**]	Lock a file.
	FOpen file_number, file_name, mode	Open a file.
	FPutFile file_number	Puts a file into the SIM file service server.
	FRead file_number, user_variable or list_spec[, user_variable or list_spec...]	Split the next line read from a file into the variables specified in the statement.
	FReadBfr file_number, data, count_to_read, count_read	Read the number of bytes specified in the command.
	FReadLn file_number, line	Read the entire line into a string variable.
	FSeek file_number, seek_position	Go to a specified position in an open file.
	FunLock file_number	Unlock a locked file.
	FWrite file_number, variable1 [, variable2][, variable3...]	Write to a formatted file.
	FWriteBfr file_number, data, count_to_write, count_written	Write a specified number of bytes.
	FWriteLn file_number, line	Write an entire line.

Category	Command and Syntax	Description
Flow Control	**Break**	Break out of the current For loop.
	Call name	Call a subroutine procedure.
	ContinueOnCancel	Continue processing the script even if the [Cancel] or [Clear] key is pressed after an Input command has been issued.
	Event Inq : number • **Tmed** : number • **RxMsg** : event_ID • **Final_Tender** : no event_ID required • **Print_Header** : alpha/numeric • **Print_Trailer** : alpha/numeric ... **EndEvent**	Indicate the start and end of an event procedure.
	ExitCancel	Exit a script and cancel the current tendering operation.
	ExitContinue	Exit a script and continue the current tendering operation.
	ExitOnCancel	Exit a script when the [Cancel] or [Clear] key is pressed after an Input command has been issued.
	ExitWithError	Display a defined error message and exit the script.
	For...**EndFor** counter = start_expression To end_expression [**Step** increment]	Perform commands a specified number of times.
	ForEver...EndFor	Perform commands an indefinite number of times.
	If…**Else**...**EndIf** expression [operator...] expression [operator...]	Execute commands if the specified condition is met.
	Return	Return from a subroutine.
	Sub...**EndSub** name **Sub**...**EndSub** name (**Ref** | **Var** parameter [, **Ref** | **Var** parameter]...)	Indicate the start and end of a subroutine procedure.

Category	Command and Syntax	Description
Flow Control continued	**While**...**EndWhile** expression	Execute a loop structure until an expression becomes FALSE.
Input/Output	**Beep**	Sound the beeper.
	ClearChkInfo	Clear check information detail lines in buffer.
	ClearIslTs	Clear any previously defined touchscreen keys.
	ClearKybdMacro	Clear macro key definitions.
	ClearRearArea	Clear the contents of the customer display.
	Display row, column, expression[{output_specifier}] [expression [{output_specifier}]...]	Display text or a field at a defined place within a window.
	DisplayInput row, column, input_variable[{input/output_specifier}, \ prompt_expression][, prompt_expression,...]	Display an input field within a window.
	DisplayInverse row, column, expression[{output_specifier}]}[, expression\ [{output_specifier}]...]	Display input field in inverse video.
	DisplayIslTs	Display an ISL-defined touchscreen.
	DisplayKBArea prompt_expression	Display data in the keyboard entry area of a Keyboard Workstation.
	DisplayMSInput row, column, input_variable}\ [{input/ output_specifier}], \ prompt_expression[, row, column,\ input_variable{input/ output_specifier},\ prompt_expression,...]	Display an input field within a window and allow magnetic card swipe to satisfy field entry.
	DisplayRearArea	Display up to 20 characters on the POS workstation customer display.
	ErrorBeep expression{output_specifier}[, expression[{output_specifier}],...]	Sound an error beep.
	ErrorMessage	Display an error message and continue.

Category	Command and Syntax	Description
Input/Output continued	**GetEnterOrClear**	Wait for the [Enter] or [Clear] key to be pressed.
	GetTime	Retrieve current time.
	Input	Capture operator entry for a single field or prompt.
	Inputkey	Capture operator entry and a key for a single field or prompt.
	ListDisplay (W)	Display a list.
	ListInput (W)	Display a list and get an operator selection.
	ListInputEx	Display a list and get an operator selection. Does not provide a WROW or WCOL variable.
	LoadDbKybdMacro	Load a pre-defined keyboard macro so that it may be executed upon successful completion of a script.
	LoadKybdMacro	Load a user-defined keyboard macro so that it may be executed upon successful completion of a script.
	PopUpIslTs	Display a touchscreen as a pop-up.
	Prompt expression[{output_specifier}] [, expression[{output_specifier}],...]	Display an operator prompt.
	SaveChkInfo expression[{output_specifier}] [, expression[{output_specifier}],...]	Insert check information detail into the check.
	ScanBarcode	Scan Barcodes that contain QR codes (more than 40 chars).
	SetIslTsKey row, col, num_rows, num_cols, font, key_expression, expression	Define a touchscreen key.
	Touchscreen numeric_expression	Activate a touchscreen for the duration of this operation.
	WaitForClear [prompt_expression[{output_specifier}] [, prompt_expression\ [{output_specifier}],...]	Wait for the [Clear] key before continuing. If no prompt text is supplied, “Press Clear to Continue” is the default.

Category	Command and Syntax	Description
Input/Output continued	**WaitForConfirm** [prompt_expression[{output_specifier}] [, prompt_expression\ [{output_specifier}],...]	Wait for an operator confirmation. If no prompt text is supplied, “Press Enter to Continue” is the default.
	WaitForEnter [prompt_expression[{output_specifier}] [, prompt_expression\ [{output_specifier}],...]	Wait for the [Enter] key before continuing. If no prompt text is supplied, “Press Enter to Continue” is the default.
	Window row, column [, expression[{output_specifier}],...]	Create a window of specified size and optionally display a window title.
	WindowClear	Clear a display window.
	WindowClose	Close the current window.
	WindowEdit[WithSave]	Display the current contents of specified variables within a window and allow them to be edited; optionally require the [Save] key to save entries and exit.
	WindowInput[WithSave]	Display the specified fields within a window, without the present contents; optionally require the [Save] key to save entries and exit.
Miscellaneous	**LowerCase**	Convert a string to lower-case.
	MSleep *milliseconds*	Sleep for the requested number of milliseconds.
	SimDB *interface_number, request_msg, response_messag*	Used by the SIM to send a request to the SIMDB DLL and then receive a response.
	System	Execute a command.
	UpperCase	Convert a string to upper-case.
	UseBackupTender	Use backup tender programmed in the Simphony database.
	WindowScrollDown	Scroll current window down one line.
	WindowScrollUp	Scroll current window up one line.

Category	Command and Syntax	Description
Printing	**FormatRaw** argument, max_size, data	This command allows a SIM script to send up to 2 Kilobytes of raw (un-altered) data to only IDN, Serial, IP, and Bluetooth printers.
	LabelFeedToPeel	Feeds printed labels to the label peeling position. Only works on the Epson L90 label printer.
	LineFeed [number_of _line_feeds]	Linefeed one or multiple lines.
	ListPrint list_size, array	Print a list.
	PrintLine expression[{output_specifier}] or directive \ [, expression[{output_specifier}] or directive...]	Print specified text and/or fields.
Variables	**ClearArray** array_variable	Clear an array
	Format string_variable [, field_sep_char] **as** expression[[{output_specifier}],\ expression[{output_specifier}],...]	Concatenate one or more variables into a string.
	FormatBuffer source_string, destination_string	Format a non-printable string into a printable string.
	FormatQ string_variable [, field_sep_char] **as** expression[{output_specifier}],\ expression[{output_specifier}],...]	Concatenate one or more variables into a string and enclose the string in quotes.
	MakeAscii source_string, destination_string	Remove any non-ASCII characters from a string.
	Mid (string_variable, start, length**) =** replacement_string	Set one portion of a string equal to another string.
	ProRate	Prorate itemizers for chg posting.
	[Retain/Discard]GlobalVar	Retain or discard global variables between transactions.
	SaveRefInfo expression[{output_specifier}] [, expression\[{output_specifier}],...]	Save information as tender/ media reference detail.
	SaveRefInfox ref_type, expression[{output_specifier}] [, expression\ [{output_specifier}],...]	Save information as tender/ media reference detail with reference type.

Category	Command and Syntax	Description
	SetReRead	Re-read the ISL script for new or changed ISL scripts.
	SetSignOn[Left/Right]	The minus sign will go on the left or right side, respectively, when formatting numbers.
Variables continued	**SetString** main_string, character_string[, count]	Replace all or a specific number of characters in a string with a particular character.
	Split string_to_split, field_sep_char, user_variable or list_spec \ [, user_variable or list_spec...]	Break a string into separate fields.
	SplitQ	Break a string into separate fields and enclose the string in quotes.
	UseSortedDetail	Consolidated detail is accessible.
	UseStdDetail	Raw detail is accessible.
	Var	Declare a variable field of specified type that will be used for input and/or used in an interface message.

Note: *The meaning of the data within the braces will be explained later.*

Functions

Command Name and Syntax
Abs (*integer* or *decimal***)**
ArraySize (*array_name***)**
Asc (*string_expression***)**
Bit (*hex_string, bit_position***)**
Chr (*integer***)**
Env (*environment_variable***)**
Feof (*file_number***)**
FTell (*file_number***)**
GetHex (*hex_string***)**
Instr (*index, string_expression, character***)**
Key (*key_pair***)**
KeyNumber (*key_expression***)**
KeyType (*key_expression***)**
Len (*string_expression***)**
Mid (*string_expression*, *start, count***)**
ToInteger (*decimal***)**
Trim (*string_expression***)**
VarSize (*user_variable***)**

Appendix D

Key Types, Codes, and Names

In This Chapter

This chapter lists the Key Types, Codes, and Names from Simphony.

Type 11 Function Key Categories

Movement Keys	
21 - End	26 - Right
22 - Down	27 - Home
23 - Page Down	28 - Up
24 - Left	29 - Page Up

NLU Keys	
101 - NLU . . . 132 - NLU	141 - Course NLU . . . 172 - Course NLU

Sales NLU Keys
200 - Discount NLU
201 - Service Charge NLU
202 - Tender/Media NLU

General Keys		
300 - Launch PMC	307 - Sign in UWS RVC 3	313 - Minimize Application
301 - Help	308 - Sign in UWS RVC 4	314 - Close Application
302 - Help, Prompt Number	309 - Sign in UWS RVC 5	315 - Enter Offline Mode
304 - Display Time	310 - Sign in UWS RVC 6	316 - Exit Offline Mode
305 - Sign in UWS RVC 1	311 - Sign in UWS RVC 7	320 - Reload Workstation Database
306 - Sign in UWS RVC 2	312 - Sign in UWS RVC 8	

Touchscreen Keys		
350 - TS No Key Display	355 - TS Shift	360 - TS Next Screen
351 - TS SLU Page Up	356 - TS Pop-up	361 - TS Previous Screen
352 - TS SLU Page Down	357 - TS Close Pop-up	362 - TS Staydown Screen
353 - TS SLU Home	358 - TS Contrast Up	363 - TS Label Only
354 - TS SLU End	359 - TS Contrast Down	

Mobile MICROS Key
382 - Select Printers

Check Begin/ Pickup Keys		
399 - Begin Party Chk	414 - Pickup, Tbl #, Rvc 1	427 - Chg Trn Rvc, Rvc 3
400 - Begin Chk by Num	415 - Pickup, Tbl #, Rvc 2	428 - Chg Trn Rvc, Rvc 4
401 - Begin Chk by Table	416 - Pickup, Tbl #, Rvc 3	429 - Chg Trn Rvc, Rvc 5
402 - Pickup by Number	417 - Pickup, Tbl #, Rvc 4	430 - Chg Trn Rvc, Rvc 6
403 - Pickup, Chk #, Rvc ?	418 - Pickup, Tbl #, Rvc 5	431 - Chg Trn Rvc, Rvc 7
404 - Pickup, Chk #, Rvc 1	419 - Pickup, Tbl #, Rvc 6	432 - Chg Trn Rvc, Rvc 8
405 - Pickup, Chk #, Rvc 2	420 - Pickup, Tbl #, Rvc 7	435 - Begin Check by ID
406 - Pickup, Chk #, Rvc 3	421 - Pickup, Tbl #, Rvc 8	436 - Pickup Check by ID
407 - Pickup, Chk #, Rvc 4	422 - Adjust Closed Check	437 - Transfr Check by ID
408 - Pickup, Chk #, Rvc 5	423 - Reopen Closed Check	438 - Guest Check ID
409 - Pickup, Chk #, Rvc 6	442 - Adjust Closed Check (Prev. Days)	439 - Pickup Check SLU
410 - Pickup, Chk #, Rvc 7	443 - Reopen Closed Check (Prev. Days)	444 - Next Drive Thru Order
411 - Pickup, Chk #, Rvc 8	424 - Chg Trn Rvc, Rvc ?	445 - Insert Order
412 - Pickup by Table	425 - Chg Trn Rvc, Rvc 1	446 - Insert Order After Last Paid

Check Begin/ Pickup Keys		
413 - Pickup, Tbl #, Rvc ?	426 - Chg Trn Rvc, Rvc 2	447 - Suspended Check SLU

Check Operations Keys		
499 - Add/Xfr Check SLU	516 - Add/Xfr by Tbl, RVC 5	548 - Edit Dlt Xfr All
500 - Add/Xfr by #	517 - Add/Xfr by Tbl, RVC 6	549 - Add Team Member
501 - Add/Xfr by #, RVC ?	518 - Add/Xfr by Tbl, RVC 7	550 - Remove Team Member
502 - Add/Xfr by #, RVC 1	519 - Add/Xfr by Tbl, RVC 8	551 - TMS Bus Table
503 - Add/Xfr by #, RVC 2	522 - Table Number	552 - TMS Clear Table
504 - Add/Xfr by #, RVC 3	523 - Number of Guests	553 - TMS Close Table
505 - Add/Xfr by #, RVC 4	524 - Print Customer Receipt	554 - TMS Xfr Tbl
506 - Add/Xfr by #, RVC 5	534 - Exempt Auto Svc Chg	555 - TMS Xfr Tbl RVC
507 - Add/Xfr by #, RVC 6	535 - Split Check	556 - TouchEdit
508 - Add/Xfr by #, RVC 7	536 - Order Type 1	557 - TouchSplit
509 - Add/Xfr by #, RVC 8	537 - Order Type 2	560 - Edit Chk All
510 - Add/Xfr by Tbl	538 - Order Type 3	561 - Edit Chk One
511 - Add/Xfr by Tbl, RVC ?	539 - Order Type 4	566 - Enter Guest Info
512 - Add/Xfr by Tbl, RVC 1	562 - Order Type 5	567 - Lock Guest Check
513 - Add/Xfr by Tbl, RVC 2	563 - Order Type 6	568 - Unlock Guest Check
514 - Add/Xfr by Tbl, RVC 3	564 - Order Type 7	569 - Reprint SVC Chit
515 - Add/Xfr by Tbl, RVC 4	565 - Order Type 8	

Transaction Keys		
600 - @/For	620 - Sub-Menu Level 6	639 - Chg Price SLvl
601 - Void	621 - Sub-Menu Level 7	640 - Percent Tender
602 - Void Check	622 - Sub-Menu Level 8	641 - MI Price Override

Transaction Keys		
603 - Transaction Void	623 - Main Menu Lvl NLU	642 - Transaction Return
604 - Return	624 - Sub-Menu Lvl NLU	643 - MI SKU Entry
605 - Transaction Cancel	625 - CCard Lookup	644 - Inquire Price
606 - Repeat Round	626 - CCard Lookup/Ask	645 - MajGroup Menu Item
607 - Main Menu Level 1	627 - CCard Recall	646 - FamGroup Menu Item
608 - Main Menu Level 2	628 - CCard Recall/Ask	647 - Hold Menu Item
609 - Main Menu Level 3	629 - CCard Authorize	648 - Dsply/Hide Cond
610 - Main Menu Level 4	630 - CCard Finalize	649 - Sign. Cap. Override
611 - Main Menu Level 5	631 - Initial Authorize	650 - KDS Rush Order
612 - Main Menu Level 6	632 - Manual Authorize	651 - KDS VIP Check
613 - Main Menu Level 7	633 - CCard Auth/Prompt	653 - Print Gift Receipt
614 - Main Menu Level 8	634 - CCard Fnlz/Prompt	654 - Inventory Inquire
615 - Sub-Menu Level 1	635 - Initl Auth/Prompt	655 - Auto Discount Toggle
616 - Sub-Menu Level 2	636 - Mnual Auth/Prompt	656 - Auto Discount Apply
617 - Sub-Menu Level 3	637 - Item Weight	657 - Auto Discount Remove
618 - Sub-Menu Level 4	638 - Chg Price MLvl	658 - Remove Coupon Discounts
619 - Sub-Menu Level 5		

Seat Keys		
700 - Seat # / Next Seat	702 - Filter Seat	704 - Add Seat to Filter
701 - View / Edit Seat	703 - Change Active Seat	707 - Toggle Seat View

Currency Keys		
720 - Currency 1	740 - Currency 21	762 - Currency 11, Ask Amt
721 - Currency 2	741 - Currency 22	763 - Currency 12, Ask Amt
722 - Currency 3	742 - Currency 23	764 - Currency 13, Ask Amt
723 - Currency 4	743 - Currency 24	765 - Currency 14, Ask Amt

Currency Keys		
724 - Currency 5	744 - Currency 25	766 - Currency 15, Ask Amt
725 - Currency 6	745 - Currency 26	767 - Currency 16, Ask Amt
726 - Currency 7	746 - Currency 27	768 - Currency 17, Ask Amt
727 - Currency 8	747 - Currency 28	769 - Currency 18, Ask Amt
728 - Currency 9	748 - Currency 29	770 - Currency 19, Ask Amt
729 - Currency 10	749 - Currency 30	771 - Currency 20, Ask Amt
730 - Currency 11	752 - Currency 1, Ask Amt	772 - Currency 21, Ask Amt
731 - Currency 12	753 - Currency 2, Ask Amt	773 - Currency 22, Ask Amt
732 - Currency 13	754 - Currency 3, Ask Amt	774 - Currency 23, Ask Amt
733 - Currency 14	755 - Currency 4, Ask Amt	775 - Currency 24, Ask Amt
734 - Currency 15	756 - Currency 5, Ask Amt	776 - Currency 25, Ask Amt
735 - Currency 16	757 - Currency 6, Ask Amt	777 - Currency 26, Ask Amt
736 - Currency 17	758 - Currency 7, Ask Amt	778 - Currency 27, Ask Amt
737 - Currency 18	759 - Currency 8, Ask Amt	779 - Currency 28, Ask Amt
738 - Currency 19	760 - Currency 9, Ask Amt	780 - Currency 29, Ask Amt
739 - Currency 20	761 - Currency 10, Ask Amt	781 - Currency 30, Ask Amt

Non-Sales Operations Keys		
830 - No Sale	839 - Assn Cash Drawr 1	842 - Inquire PMS2
833 - Clock In / Out	840 - Assn Cash Drawr 2	843 - Inquire PMS3
834 - Reprint Time Card	845 - Assign Cashier	844 - Inquire PMS4
835 - Direct Tips	846 - Download New RVC	850 - Inquire PMS5
836 - Direct Tips, Ask #	848 - Asgn Csh Drawr	851 - Inquire PMS6
837 - Indirect Tips	849 - Unasgn Csh Drawr	852 - Inquire PMS7
838 - Indirect Tips, Ask #	841 - Inquire PMS1	853 - Inquire PMS8

SIM Keys		
920 - SIM 1 Inq 1 . . . 939 - SIM 1 Inq 20	980 - SIM 4 Inq 1 . . . 999 - SIM 4 Inq 20	1040 - SIM 7 Inq 1 . . . 1059 - SIM 7 Inq 20
940 - SIM 2 Inq 1 . . . 959 - SIM 2 Inq 20	1000 - SIM 5 Inq 1 . . . 1019 - SIM 5 Inq 20	1060 - SIM 8 Inq 1 . . . 1079 - SIM 8 Inq 20
960 - SIM 3 Inq 1 . . . 979 - SIM 3 Inq 20	1020 - SIM 6 Inq 1 . . . 1039 - SIM 6 Inq 20	

Multilingual Keys		
1100 - Screen Lang 1	1104 - Screen Lang List	1107 - Print Lang 3
1101 - Screen Lang 2	1105 - Print Lang 1	1108 - Print Lang 4
1102 - Screen Lang 3	1106 - Print Lang 2	1109 - Print Lang List
1103 - Screen Lang 4		

Stored Value Cards Keys		
1200 - Issue 1	1250 - Redeem Auth 3	1306 - Issue Points 6
1201 - Activate 1	1251 - Manual Redemption 3	1307 - Redeem Points 6
1202 - Reload 1	1252 - Issue Batch 3	1308 - Point Inquire 6
1203 - Cash Out 1	1253 - Activate Batch 3	1309 - Redeem 6
1204 - Balance Inquire 1	1254 - Coupon 3	1310 - Redeem Auth 6
1205 - Balance Transfer 1	1261 - Activate 4	1311 - Manual Redemption 6
1206 - Issue Points 1	1262 - Reload 4	1312 - Issue Batch 6
1207 - Redeem Points 1	1263 - Cash Out 4	1313 - Activate Batch 6

Stored Value Cards Keys		
1208 - Point Inquire 1	1264 - Balance Inquire 4	1314 - Coupon 6
1209 - Redeem 1	1265 - Balance Transfer 4	1320 - Issue 7
1210 - Redeem Auth 1	1266 - Issue Points 4	1321 - Activate 7
1211 - Manual Redemption 1	1267 - Redeem Points 4	1322 - Reload 7
1212 - Issue Batch 1	1268 - Point Inquire 4	1323 - Cash Out 7
1213 - Activate Batch 1	1269 - Redeem 4	1324 - Balance Inquire 7
1214 - Coupon 1	1270 - Redeem Auth 4	1325 - Balance Transfer 7
1220 - Issue 2	1271 - Manual Redemption 4	1326 - Issue Points 7
1221 - Activate 2	1272 - Issue Batch 4	1327 - Redeem Points 7
1222 - Reload 2	1273 - Activate Batch 4	1328 - Point Inquire 7
1223 - Cash Out 2	1274 - Coupon 4	1329 - Redeem 7
1224 - Balance Inquire 2	1280 - Issue 5	1330 - Redeem Auth 7
1225 - Balance Transfer 2	1281 - Activate 5	1331 - Manual Redemption 7
1226 - Issue Points 2	1282 - Reload 5	1332 - Issue Batch 7
1227 - Redeem Points 2	1283 - Cash Out 5	1333 - Activate Batch 7
1228 - Point Inquire 2	1284 - Balance Inquire 5	1334 - Coupon 7
1229 - Redeem 2	1285 - Balance Transfer 5	1340 - Issue 8
1230 - Redeem Auth 2	1286 - Issue Points 5	1341 - Activate 8
1231 - Manual Redemption 2	1287 - Redeem Points 5	1342 - Reload 8
1232 - Issue Batch 2	1288 - Point Inquire 5	1343 - Cash Out 8
1233 - Activate Batch 2	1289 - Redeem 5	1344 - Balance Inquire 8
1234 - Coupon 2	1290 - Redeem Auth 5	1345 - Balance Transfer 8
1240 - Issue 3	1291 - Manual Redemption 5	1346 - Issue Points 8
1241 - Activate 3	1292 - Issue Batch 5	13047 - Redeem Points 8
1242 - Reload 3	1293 - Activate Batch 5	1348 - Point Inquire 8
1243 - Cash Out 3	1294 - Coupon 5	1349 - Redeem 8
1244 - Balance Inquire 3	1300 - Issue 6	1350 - Redeem Auth 8
1245 - Balance Transfer 3	1301 - Activate 6	1351 - Manual Redemption 8

Stored Value Cards Keys		
1246 - Issue Points 3	1302 - Reload 6	1352 - Issue Batch 8
1247 - Redeem Points 3	1303 - Cash Out 6	1353 - Activate Batch 8
1248 - Point Inquire 3	1304 - Balance Inquire 6	1354 - Coupon 8
1249 - Redeem 3	1305 - Balance Transfer 6	

Tax Exempt/Shift Keys		
525 - Exempt All Taxes	540 - Tax Shift Rate 1	1411 - Tax Shift Rate 9
526 - Exempt Tax Rate 1 . . . 533 - Exempt Tax Rate 8	. . . 547 - Tax Shift Rate 8	. . . 1466 - Tax Shift Rate 64

Type 9 Keypad Keys

Keypad Keys		
0 - Numeric “0” key	8 - Numeric “8” key	16 - Exit PCWS App
1 - Numeric “1” key	9 - Numeric “9” key	17 - Add
2 - Numeric “2” key	10 - Numeric “00” key	18 - Delete
3 - Numeric “3” key	11 - Decimal “.” key	19 - Edit
4 - Numeric “4” key	12 - Enter	20 - Edit Delete
5 - Numeric “5” key	13 - Clear	21 - Edit Insert Tggl
6 - Numeric “6” key	14 - Shift	22 - Current MMDDYY
7 - Numeric “7” key	15 - Backspace	

Appendix E

sendsim

In This Chapter

This chapter describes the sendsim program. Note that sendsim is not currently supported in Simphony.

sendsim

The sendsim program is the SIM utility that provides customized messaging and paging services via MICROS workstations, including Mobile MICROS, from the Windows® command line.

Note: *sendsim is not currently supported in Simphony.*

The sendsim user should already be familiar with SIM concepts and ISL programming.

Requirements

This SIM utility requires two parts:

- The sendsim program
- An ISL script, which responds to the messages generated by sendsim

Syntax

The following is the complete sendsim command line syntax:

```
sendsim ws# pms# msgtype [msg arg] [msg arg] [msg arg]...
```

Each command line argument is described in the table below:

Argument	Description	Comment(s)
ws#	workstation number that the message should be sent to	Use 0 if the message should go to all workstations.
pms#	PMS definition object number associated with this event	Since SIM script files executed by OPS are associated with a PMS record, the PMS number must be specified so that OPS can determine the script to execute (this value must be non-0 to have meaning).
msgtype	message type[a]	This corresponds to the RxMsg event type in an ISL event.

Argument	Description	Comment(s)
[*msg arg*]	optional message arguments	These can be read from ISL using the RxMsg command. Any number of arguments can be sent, and their contents are not examined by sendsim.

a. The message type is not case-sensitive.

Operations

Each user workstation is controlled by an individual Windows process. If there are 20 terminals, then 20 processes (called OPS) are running on the Windows PC. Whenever, an OPS process is waiting for input from the user, it will also wait for data from the sendsim program.

When an OPS process receives a sendsim message, it will attempt to run an ISL RxMsg event whose event ID corresponds to the msgtype parameter in the command line. If the command line were:

```
sendsim 0 1 mymessage
```

OPS would try to run the ISL script event:

```
event RxMsg : mymessage
```

Once the event is found, normal ISL processing occurs. The RxMsg command should be used to read any additional arguments into local variables.

Remember the following when running sendsim:

- The sendsim program sends the message, but does not wait for a response from all OPS processes. There is no way to send data back to the sendsim program.
- It is possible to run the sendsim program when OPS is not running, but all messages are lost.

- If the ISL script file is not present, or the message type is not present in the ISL script file, no error is generated. Ops will throw away the message. In Example 2 on page FE-5, if the first line was written as

  ```
  event RxMsg : mymsg
  ```

  then Ops would throw away the received message since there was no ISL event for the mymessage event, only for the mymsg event.

- If OPS is already processing a sendsim message, it will queue the message for subsequent processing.

Examples

Example 1

The following example shows the sendsim program being used to send a message to all OPS processes from the command line, and the ISL script file needed to receive the data and display it on the workstation display.

COMMAND LINE:

```
sendsim 0 1 mymessage "We are out of broiled flounder"
```

Each command line argument is defined in the table below:

Argument	Character(s)	Comment(s)
ws#	0	Send message to all workstations.
pms#	1	A PMS link in the *RVC Parameters* module must use this object number.
msgtype	mymessage	Used by the ISL script event.
[*msg arg*]	“We are out of broiled flounder”	This message[a] MUST be enclosed in quotes to be interpreted as one argument. Without the quotes, each word would become a separate argument, for a total of six arguments.

a. The message argument can be a text string or any command line entry compatible with the user’s shell.

ISL SCRIPT:

Since the message is being sent to PMS number 1, the script file is named pms1.isl, and placed in the `/micros/simphony/etc` directory. The script contents are:

```
event RxMsg : mymessage
    var data:A60                // variable to contain message
    RxMsg data                  // get first argument
    window 3, len(data)+2       // create window
    display 2, @center, data    // display message
    waitforclear                // wait for clear key
endevent
```

Note: *This event may pop up at any point in the workstation transaction.*

Example 2

If adding the following event to the ISL script, any user at a terminal can send a message to another terminal.

```
event inq:1
    var msg:A40, ws:N9
    input ws, "Enter workstation number"
    input msg, "Enter message"
    system "sendsim ", ws, " ", @pmsnumber, " mymessage ", msg{"}
endevent
```

Note: *The use of the system command to run the sendsim program.*

Troubleshooting

If the sendsim program is run, but the OPS process(es) does not receive the message, then check the following:

- Is the system up and OPS running? If no Ops processes are running, then no one can receive the message.
- Is the PMS object number valid? That is, does it exist in the database with valid field information?
- Is there a link to the PMS in the *RVC Parameters* module?
- Is the command line syntax in the correct order? Type **`sendsim`** without any arguments to view the command line syntax.
- If the workstation number is not 0, does it refer to the number in the workstation table and not the device table?
- Is there a valid SIM script in the `/micros/simphony/etc` directory? Does it have the proper EVENT RxMsg event?
- Is a SIM script already running on the workstation that will be receiving the message? If so, the message will be queued for later processing.

Appendix F

Windows DLL Access

In This Chapter

This chapter describes Windows DLLs and how to access them using SIM.

Windows DLL Access

Overview

This document describes DLLs and how to access them using SIM. This information is broken down into the following sections:

- What is a DLL?
- Using DLLs
- Simphony SIM DLL Support
- Using Simphony SIM DLL Commands

What is a DLL?

The Simphony System Interface Module (SIM) can use Windows® Dynamic Link Libraries (DLLs). DLLs are modules that contain functions and data. They provide a way of separating applications into small manageable pieces that can be easily modified and reused.

Dynamic Linking

Dynamic linking provides a means of giving applications access to function libraries at run-time. DLLs are not copied into an application's executable files. Instead, they are linked to an application when it is loaded and executed.

Dynamic link libraries reside in their own separate files. Applications load them into memory when they are needed, and share a single copy of the DLL code in physical memory. A single DLL can be used by several applications simultaneously. This in turn saves memory and reduces swapping.

Windows allows only a single instance of a DLL to be loaded into memory at any time. When a DLL is being loaded, Windows checks all the modules already in memory. If it doesn't find a match, then it loads the DLL. If it does find a match, and the matching module is a DLL, it doesn't load it again.

The following are steps that an application takes when calling a function in a DLL:

- The application uses the `LoadLibrary` or `LoadLibraryEx` function to load the DLL at run-time. This, for example, can be a .DLL or .EXE.
- The application calls the `GetProcAddress` function to get the addresses of the exported DLL functions and it is mapped into the address space of the calling process.
- The application then calls the exported DLL functions using the function pointers returned by `GetProcAddress`.

DLLs can define two kinds of functions: exported and internal. The exported functions can be called by other applications. Internal functions can only be called from within the DLL where they are defined. DLLs can contain code, data, and resources such as bitmaps, icons, and cursors. These are stored as executable programs.

Using DLLs

DLLs are used for three general purposes:

- Sharing components
- Encapsulating (hiding) data and code
- Performing system-level operations

Sharing Components

DLLs provide an easy way for multiple applications to share components. These components can be:

- Code: a DLL provides one copy of its code for all applications that need it.
- Data: by storing and retrieving data, applications can communicate with each other. The DLL provides a function for applications to store and retrieve data in its data segment.
- Custom Controls: these can be placed in DLLs for use by multiple applications. They can be written by developers and marketed as separate DLLs or used in applications.
- Resources: icons, bitmaps, fonts, and cursors can be placed in DLLs. Device drivers are also DLLs that provide system resources.

Encapsulating (hiding) Data and Code

DLLs can be used to hide data and code. A DLL can implement an abstract data type (ADT), which can be used by applications. The applications can use the ADT without knowing anything about the actual implementation. When changes are made to the data structures and internal code, the applications that use the DLL don't have to be modified or recompiled.

Performing System-level Operations

DLLs can be used to perform low-level operations. Operations such as interrupt service routines can be placed in fixed-code segments of DLLs. If an application needs to issue interrupts, the code can be placed in a DLL rather than in the application. Also, a DLL can be written as a device driver for certain pieces of hardware (e.g., a mouse or keyboard).

Simphony SIM DLL Support

The Simphony System Interface Module allows the programmer to reap the benefits of DLLs. The programmer can write or use existing DLLs to further enhance the capabilities of the Simphony POS Operations module. Using three SIM commands, the programmer can create a SIM script to access an externally created DLL. This functionality considerably broadens the scope of the Simphony SIM feature. Some advantages of this feature are:

- The ability to write SIM scripts to customize the Simphony PMS interface.
- The opportunity to take advantage of third-party development.
- A wider range of creativity.
- Faster turn around time.

Customization

Using DLLs allows the flexibility of customization. Now the programmer is no longer confined to the text based window that SIM uses for input and output. For example, one could create a custom interface to a PMS that returns guest information to the Simphony POS Operations module.

Third-party Development

If the programmer is not interested in writing DLLs, one can take advantage of DLLs that can be, or already have been, written by a third party for Simphony SIM.

Creativity

Using the resources available with DLLs, fonts, bitmaps, cursors, etc. can be used to create appealing user interfaces.

Faster Turn Around Time

Using DLLs can increase the turn around time of certain system requests. For example, a DLL could be created to connect to the Simphony database and perform a custom query on guest check information.

DLLs allow virtually unlimited flexibility when creating scripts to enhance the functionality of the POS Operations module.

Using Simphony SIM DLL Commands

Three commands allow SIM to access DLLs. They enable a SIM script to call an externally created DLL. These three commands are:

- DLLLoad: loads the external DLL
- DLLCall: calls a function contained in the DLL
- DLLCallW: calls a function contained in the DLL with Unicode
- DLLFree: frees a loaded DLL

DLLLoad

The `DLLLoad` command is used to load the external DLL. It needs to be called only once during the lifetime of the SIM script. The syntax is:

```
DLLLoad handle, name
```

where `handle` is a SIM N9 variable, and `name` is a SIM string expression. The 'name' parameter is used to identify the DLL, and the resulting handle is stored in the 'handle' variable. An example of this would be:

```
var dll_handle:N9
DLLLoad dll_handle, "myops.dll"
```

If the command fails, then `dll_handle` will be 0. If not, then `dll_handle` is the handle used for any further accesses for the other DLL functions.

This function is just a wrapper around the `Windows LoadLibrary()` function. All the rules which apply to path settings apply here.

If the DLL is already loaded, then Windows will prevent it from being loaded again.

DLLCall

The `DLLCall` command is used to call a function contained in the DLL. The syntax for this command is:

```
DLLCall handle, dll_name( [parm1 [parm2 [parm3...]]] )
```

where `handle` is the previously loaded library handle, `dll_name` is the name of the function, and `parm#` are the optional parameters.

This command performs two tasks:

- Get the address of the function using the Windows `GetProcAddress()` function. If this function fails, then an ISL error is generated.
- Call the DLL function.

Since it is not possible for SIM to check the validity of the parameters, it is up to the SIM script writer to ensure that the proper number and type is used.

Note: *It is not possible for a DLL to return a value back to SIM. More specifically, if one is returned, it is ignored. Passing information back to the SIM script should be done using references.**Refer to the Parameter Passing section on page F-7.*

DLLCallW

The `DLLCallW` command is used for string variables. The value is interpreted as Unicode data when passed back into SIM. This command calls a function contained in the DLL with Unicode. The syntax for this command is:

```
DLLCallW handle, dll_name( [parm1 [parm2 [parm3...]]] )
```

where `handle` is the previously loaded library handle, `dll_name` is the name of the function, and `parm#` are the optional parameters.

This command performs the same tasks as DLLCall.

Since it is not possible for SIM to check the validity of the parameters, it is up to the SIM script writer to ensure that the proper number and type is used.

DLLFree

The `DLLFree` command is used to free a loaded DLL. The syntax is:

```
DLLFree handle
```

where `handle` is the handle obtained in a `DLLLoad` command. This function is automatically called when OPS exits, or the SIM script is reread. It can be called to free up resources loaded by the DLL.

Parameter Passing

There are three types of variables in SIM:

- integers
- strings
- monetary data

Only these types may be used as parameters to the DLL. However, each type may be passed in by value or by reference, and each type may also be passed in as an array. Therefore, there are 12 possible parameter types. Refer to the table on page F-8.

To pass by reference, place the string “ref” before the parameter. For example:

```
// Pass by value
DLLCall handle, my_function( count )
// Pass by reference
DLLCall handle, my_function( ref count )
```

Only variables can be passed in by reference. Complex expressions must be passed in by value.

```
// Will generate an error
DLLCall handle, my_function( ref count + 1 )
DLLCall handle, my_function( ref (1 + 3 ) )
```

When generating the parameter list, SIM will dynamically allocate data for each parameter, and this allocated data (or a reference to it) will be passed to the DLL. No references to the stored copy of the data will be passed. If by reference, SIM will copy in the new values to the data variables once the function has completed. Therefore, do not pass in huge arrays by reference when not needed, since SIM will attempt to copy in each array element, even if it has not changed.

All pointers passed to the DLL are passed in as 32-bit pointers. All strings are C nul-terminated strings. All integer values are signed. All arrays are passed as a pointer to a list of pointers or integers. The length of the array must either be passed in as an argument, or known prior by the DLL.

Though there are 12 ways of passing in parameters, there are only five separate ways to declare them in C.

1	int a	// integer
2	int *a	// integer pointer
3	int a[]	// array of integers

4	char *a
	-or-
	wchar_t *a
		// string pointer
		-or-
		// string pointers (use for Unicode)

5	char *a[]
	-or-
	wchar_t *a[]
		// array of string pointers
		-or-
		// array of string pointers (use for Unicode)

The following table lists the different possibilities and how they map to the parameter type.

SIM Type	C Declaration	Comments
integer	int a	
integer by reference	int *a	
integer array	int a []	
integer array by reference	int a []	same as integer array
string	char *a	
string reference	char *a	same as string
string array	char *a []	
string array by reference	char *a []	same as string array
amount	char *a	
amount by reference	char *a	same as amount
amount array	char *a	
amount array by reference	char *a	same as amount array
integer array	int a []	

Integers

By value

N1-N9 integers are passed in as 32-bit signed values. N10 and above are passed in as pointers to a string which contains the numeric value expressed as a string.

For example, an N12 numeric variable whose value is 12345 will be passed in as "12345." If the number is negative, then a "-" will be the first character in the string. (The reason for this is that a 32-bit integer can only have nine digits.)

By value examples

```
var a:N9 = 4
var b:N10 = 3012108000
DLLCall handle, my_function( a, 10, (1 + a) * 10, b )
```

The DLL prototype should look as follows:

```
void my_function( int p1, int p2, int p3, char *p4 );
```

The DLL function should expect these parameters:

4
10
50
"3012108000"

By reference

N1-N9 integers are passed in as pointers to a 32-bit value. The DLL can change this value, and this change will be reflected in the variable.

N10 variables are passed in as strings. The length of the string is guaranteed to be the length of the string variable.

An N12 variable with a value of "123" will be passed in as "123," but the space occupied by the string will be at least 13. This means that the DLL can safely copy in a string longer than "123."

By reference examples

```
var a:N9 = 12
var b:N12 = 3012108000
DLLCall handle, my_function( ref a, ref b )
```

The DLL prototype should look like:

```
void my_function( int *p1, char *p2 )
```

The dll function should expect these parameters:

pointer to 12
“3012108000”

Strings

All string parameters are passed in as a pointer. When by reference, SIM will copy the string data back into the original variable.

By value

The string parameter is passed in as a pointer to a nul-terminated string.

By value examples

```
var a:A20 = “12345”
DLLCall handle, my_function( a, “hello” )
```

The DLL prototype should look like:

```
void my_function( char *p1, char *p2 )
```

The DLL function should expect these parameters:

“12345”
“hello”

By reference

The string parameter is passed in as a pointer to a nul-terminated string. However, the memory reference will be guaranteed to have allocated enough space for the declared string.

For example, if a string is declared as A20 but is set to “12345,” then the string passed in will be “12345,” but will have 20 characters allocated (not including the nul terminator) The DLL can then write up to 20 characters into the string.

By reference example

```
var a:A20
DLLCall handle, my_function( ref a )
```

The DLL prototype should look as follows:

```
void my_function( char *p1 )
```

The DLL should expect these parameters:

“”””

The DLL could copy a string of up to 20 characters.

```
memset( p1, '-', 20 )
p1[ 20 ] = 0
```

Monetary Data

All SIM monetary data ($ variables) are kept as strings internally. Each string consists of the digits which make up the value. There is no representational difference between monetary amounts and numeric data N10 and greater.

The difference between the two is determined by the operations allowed on the values. The operations involved are the assignment and arithmetic ones. Therefore, monetary amounts are passed to the DLL as numeric strings. However, they will have the decimal point inserted into the proper place.

By value

The string parameter is passed in as a pointer to a nul-terminated string. A ‘-’ will be placed at the beginning of the string if it is negative.

By value example

```
var a:$12 = 12.34
DLLCall handle, my_function( a, a + 1.11, 5.67 )
```

The DLL prototype should look as follows:

```
void my_function( char *p1, char *p2, char *p3 )
```

The DLL should expect these parameters:

“12.34”
“13.45”
“5.67”

By reference

The string parameter is passed in as a pointer to a 0-padded nul-terminated string. The string will be padded with as many zeroes to make it the same as its declared length. A $12 variable string will have 12 digits and one decimal point. (The nul is not included.)

By reference example

```
var a:$12 = 12.34
DLLCall handle, my_function( ref a )
```

The DLL prototype should look like:

```
void my_function( char *p1 )
```

The DLL should expect these parameters:

"0000000012.34"

Array References

All arrays are passed in as an array of pointers or integers. As with SIM subroutine calls, all arrays must have a [] following them.

Note that arrays and arrays by reference use the same declaration. The only difference between the two is that the values are copied back to the variables when the function is done.

N1-N9 numeric values are passed in as integers.

```
var array[ 20 ] : N9

// Passing in by value.
// DLL prototype should be: void my_function( int array[] );
DLLCall handle, my_function( array[] )

// Passing in by reference.
// DLL prototype should be: void my_function( int *array[] );
DLLCall handle, my_function( ref array[] )
```

All other types are passed in as strings.

```
var array1[ 20 ] : $12
var array2[ 40 ] : A40

// Passing in by value
// DLL prototype should be:
// void my_function( char *array1[], char *array2[] );
DLLCall handle, my_function( array1[], array2[] )

// Passing in by reference
// DLL prototype should be:
// void my_function( char *array1[], char *array2[] );
DLLCall handle, my_function( ref array1[], ref array2[] )
```

DLL Error Messages

The following is a list of DLL error messages and an example of each.

Note: *If the DLL itself has an error in the function, or the wrong values are passed into it, the Ops process of the workstation running the SIM application will fail, and a signal 11 error will display on the server.*

To correct this problem, restart the workstation from the System/Control Workstations module within the Enterprise Management Console (EMC).

General Error Messages

ISL error on line ###:## Can not evaluate (END OF LINE)

This error occurs when the ISL script has no DLL handle in the call line.

(###:## represents line number:column number)

Example: `DLLLoad` (no handle specified)

ISL error on line ###:## Expected operand (END OF LINE)

This error occurs when a file name is missing from the expression.

(###:## represents line number:column number)

Example: `DLLLoad dll_handle_1,`

ISL error on line ###:## File name too long

This error occurs when the file name is too long.

(###:## represents the line number:column number)

Example: `thisfilenameistoolong.dll`

ISL error on line ###:## Not a variable ("filename. dll")

This error occurs when a non-existant DLL is called.

(###:## represents line number:column number)

Example: `DLLLoad "filename.dll"`

where "filename" is the name of the DLL to be loaded, and it cannot be found

- OR -

This error occurs when the arguments are reveresed.

(### represents the line number)

Incorrect—Example: `DLLLoad "filename.dll", dll_handle_1`
Correct—Example: `DLLoad dll_handle_1, "filename.dll"`

where "filename" is the name of the DLL

Maximum number of DLLs loaded

This error occurs when the maximum number of DLLs is loaded. The maximum of 20 DLLs is allowed per SIM interface.

DLLCall Error Messages

ISL error Expected Operand

This error occurs when a DLLCall is made without arguments.

Example: `DLLCall`

ISL error Variable undefined

This error occurs when a DLLCall is made without a function name.

Example: `DLLCalldll_handle_1`

- OR -

This error occurs when a DLLCall is made with a non-existant function.

ISL error Undefined Function

This error occurs when a DLLCall is made without a handle name.

Example: `DLLCall my_function(3)`

-OR-

This error occurs when a DLLCall is made with the arguments reverseds.

Example: `DLLCall my_function(3), dll_handle_1`

DLLFree Error Messages

ISL error Expected Operand

This error occurs when a DLLFree is made without arguments.

Example: `DLLFree`

ISL error Variable undefined

This error occurs when a DLLFree is made with an incorrect handle name.

Example: `DLLFree wrong_handle_name`

Appendix G

SIM Events

In This Chapter

This chapter provides a brief overview of SIM events and how they can be used.

Overview

SIM events can be categorized as

- Directly triggered by a keystroke (e.g., INQ and TMED)
- Indirectly triggered by an event in Ops (e.g., SIGN_IN, FINAL_TENDER)
- Triggered by a SIM script (e.g., TIMER, RXMSG)

Events Directly Triggered by a Keystroke

Some characteristics of these types of events are:

- The event must be defined in the script. If a **SIM Inquire** key is pressed, then the corresponding "event inq:#" code must be in the SIM script. If the event is not in the SIM script, an error is generated.
- The event is specific to a PMS interface, and therefore, only one SIM event is executed.

Events Indirectly Triggered by an Ops Event

Some characteristics of these types of events are:

- The event doesn't have to be defined in the script. For example, if the SIM script has the FINAL_TENDER event programmed, then it will be executed whenever a check is completely tendered. If the event is *not* defined, then no error will occur.
- These events occur for *all* interfaces. When a check is completely tendered, then there will be an attempt to run the FINAL_TENDER in *all* SIM scripts. Each script has the chance to "hook in" to the event.

  Scripts are executed in the order that the PMS interface definitions appear in the Remote Management Console, in *RVC Information* | *RVC Parameters* | *Interfaces*. If the script file does *not* have the particular event, it is ignored—the events cannot be cancelled. That is running the *ExitCancel* command inside the script will not stop the operation from occurring.
- Some events have two variations—a *regular* event and a *confirm* event. The *confirm* event can optionally prevent the action from continuing, whereas the *regular* SIM cannot. Refer to page G-7 for details on confirm events.

Event-Specific Variables

Many of the events have system variables (i.e., @ variables) that are specific to that event only.

Quick Reference Table

The table below lists the events available in SIM, and the variables, if any, that can be used. Each column is described as follows:

- **T** - Identifies the type of event:
    - **K** - keystroke event
    - **O** - Ops event
    - **S** - SIM-triggered event
- **C** - Indicates the event can be a *confirm* event and the operation can be cancelled by the script (**Y**es/**N**o)
- **Event** - Identifies the name of the event in the SIM script
- **Description** - Describes what the event does
- **Variables** - Lists the specific variables (if any) that can be used with the event type

T	C	Event	Description	Variables
O	Y	ADJUST_CHECK	adjust closed check operation	---
O	Y	BEGIN_CHECK	check has been begun	---
O	Y	CLOCK_IN	employee is clocking in	---
O	Y	CLOCK_OUT	employee is clocking out	---
O	Y	CLOSE_CHECK	check has been closed (paid in full); functionality is equivalent to the FINAL_TENDER event	---
O	Y	CSH_DRWR_CLS	Ops has just required the operator to close the cash drawer on the workstation	---
O	Y	CSH_DRWR_OPN	Ops is opening the cash drawer on the workstation (this event is NOT fired if the cash drawer is somehow opened on the workstation)	---

T	C	Event	Description	Variables
O	Y	DSC	discount has been entered	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	DSC_VOID	discount has been voided	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	N	ERR_MSG	error message has occurred	@errormessage - error message
O	N	EXIT	Ops has exited	---
O	N	FINAL_TENDER	check has been paid in full	---
O	N	INIT	Ops has started	---
K	N	INQ	inquire event triggered by an **Inquire** key	---
O	N	LDS_BPS_OFF	LDS bypass has been deactivated at the workstation (for North American LDS only)	---
O	N	LDS_BPS_ON	LDS bypass has been activated at the workstation (for North American LDS only)	---
O	Y	MGR_PROC	employee is running a Manager Procedure	@mngrprocnum - manager procedure number
O	Y	MI	menu item has been ordered	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	MI_RETURN	menu item has been returned	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	MI_VOID	menu item has been voided	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	NO_SALE	No Sale key has been pressed	---
O	Y	PICKUP_CHECK	check has been picked up	---
O	Y	PICKUP_LOAN	triggered when a Pickup or Loan is performed	---
O	Y	REOPEN_CHECK	check has been reopened	---

T	C	Event	Description	Variables
O	Y	RPT_GEN	employee is running a report	@autoseq
S	N	RXMSG	message has been received from the PMS system	---
O	Y	SIGN_IN	employee is signing in	---
O	Y	SIGN_OUT	employee is signing out	---
O	Y	SRVC_TOTAL	transaction has been service totalled	---
O	Y	SVC	service charge has been entered	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	SVC_VOID	service charge has been voided	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	SYS_AUTH	authorization is required	@authemp - authorizing employee @authtype - authorization type
S	N	TIMER	an event programmed by a SIM script to fire periodically	---
K	N	TMED	tender media event triggered by a tender media key	---
O	Y	TNDR	tender has been entered	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	TNDR_VOID	tender has been voided	@obj - object number of item @qty - quantity of item @ttl - total amount of item
O	Y	TRANS_CANCEL	transaction has been cancelled	---
O	Y	VOID_CHECK	check has been voided	---
O	N	WS_RESTART	workstation has been restarted, either due to a workstation reboot or a NetCC restart	---
O	N	WS_EXIT	workstation is no longer active (e.g., a workstation may have exited to DOS)	---

SIM Confirm Events

The purpose of SIM confirm events is to allow script writers to stop certain POS operations. For example, if a third-party inventory control system is used to count menu items, then the SIM script could query the system when a menu item is ordered, and "cancel" the operation if the item is out of stock.

All confirm events will be run *before* the operation takes place, and the normal event will run after the operation takes place. Therefore, there could be two events in the script: one *confirm* and one *normal*.

Consider the following script:

```
event mi : confirm
   waitforconfirm "Press enter to order item"
endevent

event mi
   waitforclear "You have entered menu item " , @obj
endevent
```

When a menu item key is first pressed, the confirm event will be run first. If the user presses the **Clear** key, the ordering of the menu item will be cancelled. If **Enter** is pressed, the item will be ordered, and the normal event will be run afterwards.

Note: *One important point about confirm events is that, like the normal events, each SIM script will have a chance at running it. That means that each SIM script will be able to cancel the operation. It should not be assumed that allowing an operation in a confirm event will result in the operation taking place.*

In addition, a particular item cannot be specified. The events occur for all items (menu items, discounts, tenders, etc.).

Glossary

Argument

An **argument** is a generic term for an item or group of items that is used in the syntax of a command, that refers to the information that is required by the command. It may be an alphanumeric character, group of characters, or word(s) that receive the action of a command or function. For example, the **Call** command requires an argument (i.e., the variable name) in order to work.

Also see: Array, Constant, Equation, Expression, Function, Input Expression, Prompt Expression, String Expression, Syntax, System Variable, and User Variable

Array

An **array** is a set of values, based on the name of a User Variable. A User Variable Array (or Array Variable) identifies each value by the variable name and the index number, in brackets. For example, an Array called Rooms that has 20 values would be identified from Rooms [1] to Rooms [20].

Also see: User Variable

Asynchronous Serial Interface

An **asynchronous serial interface** is a full duplex interface supporting transmission speeds of 300 to 9600 baud.

Concatenate

Concatenate means to join two or more text strings together to form a single contiguous string.

Constant

A **constant** is a value that does not change (the opposite of a variable). For example, the **Window** command can use a constant (i.e., window 5, 36).

Also see: Expression and Variable

Encryption

The ISL **Encryption** Program is a utility that converts a script into an encrypted (hexadecimal) format. It is used as a security precaution for proprietary information. See “The documentation should be located in an ASCII-formatted file with the name x...x.doc, where x...x is the same as the script. For example, the readme.doc file for pms1.isl should be named pms1.doc. The readme.doc file should be placed in the \Micros\Simphony\etc directory.” on page 3-15 for a complete description.

Equation

An **equation** is a mathematical formula. The ISL may use the following operators within a formula: addition (+), subtraction (-), division (/), multiplications (*), greater than (>), or less than (<). Parentheses may be used to isolate parts of the equation, as necessary.

Also see: Expression, Formula, and Operator

Expression

An **expression** is a place holder argument that can be one of the following:

User Variable
System Variable
Constant
String
Function
Equation

Also see: Argument, Hex Expression, Input Expression, Numeric Expression, and String Expression

Format Specifiers

See Operators

Formula

A **formula** can be used to calculate numeric values, compare one value to another, and select an action based on a comparison, and join multiple string expressions into a single string. The ISL may use the following operators within a formula: addition (+), subtraction (-), division (/), multiplications (*), greater than (>), or less than (<). Parentheses may be used to isolate parts of the equation, as necessary.

Also see: Equation and Operator

Function

A **function** is a built-in ISL procedure used to evaluate fields, make calculations, or convert data.

Also see: Expression

Global Command

A **global command** is a command that is allowed outside of an event procedure. They are initialized at the beginning of each script and then maintained for the duration of that script. The following ISL commands are global:

ContinueOnCancel
DiscardGlobalVar
ExitOnCancel
Prorate
RetainGlobalVar
SetSignOnLeft
SetSignOnRight
UseBackUpTender
UseCompatFormat
UseISLFormat
UseISLTimeOuts
UseSTDTimeOuts
Var

Also see: Local Command

Global Variable

A **global variable** is declared outside an event procedure, and usually initialized at the beginning of each script. The value of each global variable is maintained throughout all Events, unless the script is changed or the **DiscardGlobalVar** command is used.

Also see: Local Command

Hex Expression

A **hex expression** is a variable or function whose value must be a hexadecimal number. This variable is used with the **GetHex** and **Bit** functions.

Also see: Expression

Input Expression

An **input expression** is an Array or User Variable that requires user input. The input expression is used by the **DisplayInput**, **DisplayMS**, **Input**, and **InputKey** commands.

Also see: Array, Expression, and User Variable

Integer

An **integer** is a positive or negative whole number or a zero, always without decimal places.

Interface Script Language (ISL)

The **Interface Script Language (ISL)** provides the facility to direct operator prompting, message formats, printing, and subsequent POS processing. A script is analyzed and executed by the System Interface Module (SIM).

Also see: ISL Script

ISL

See Interface Script Language.

Language Element

Language Elements are indivisible pieces of information which, if broken apart with whitespace, will generate an ISL error. The following items are considered language elements:

Command Names
Function Names
System Variables
Relational and Boolean Operators
Input and Output Specifiers
Comment Symbols (//)
Continuation Line Symbol (\)
Commas
Any Word and/or Symbol required by the Syntax

Local Command

Most ISL commands are considered **local commands**, in that they must be placed inside an Event procedure and only affect the processing within that event.

Also see: Global Command

Local Variable

A **local variable** must always be declared inside an event procedure, and will only be used by the event and any subroutines called by that event. Local variables are purged after each event is complete, (when an **EndEvent** is executed).

Also see: Global Variable

Nesting

Nesting is the act of using an **If**, **For**, **Forever**, or **While** command inside another. Since each of these commands is executed until its corresponding **EndIf**, **EndFor**, or **EndWhile** command is found, the entire **If**...**EndIf**, **For**...**EndFor**, **ForEver**...**EndFor**, or **While**...**EndWhile** nest must exist before the outer **End**... command. Nesting also refers to the ability to call a subroutine from within another subroutine.

Null String

A **null string** (" ") is a string expression that contains no characters. All string variables are initially set to null at the beginning of each Event procedure.

Also see: Expression and Variable

Number of Records

Number of Records is used to send and receive variable amounts of data via a list or a list array.

Numeric Expression

A **numeric expression** is a variable or function whose value must be a number. A number expression is used when specifying a touchscreen number within the Simphony database.

Also see: Expression

Offset

An **offset** is a decimal integer that is used to calculate a position of a field within a string. For example, this may be used to extract certain field information from a credit card.

Operator

An **operator** is a mathematical symbol that determines what action is taken on variables or constants in the equation. For a complete list of operators, please see “Relational and Logical Operators” on page 4-5.

Also see: Expression and String Expression

Script

A **Script** contains a series of commands, functions, and arguments that perform a particular task at the workstation and/or the PMS.

SIM

See System Interface Module.

String

A **string** is a series of connected characters (letters, numbers, symbols, spaces) stored and used as text. In ISL, a string is always in quotes.

Also see: Subroutine and User Variable

String Expression

A **string expression** is a variable or function whose value must be a string.

Also see: Expression

Subexpression

A **subexpression** is an expression within an expression. Subexpressions are used with binary operators. For example in the following expression: a + (b + c) + d, (b + c) is a subexpression.

Also see: Expression and Operators

Subroutine

A **subroutine** allows common code to be used by multiple events. Each subroutine has a unique name which is used to define it within the script, outside any event procedure. Use the **Call** command to execute a subroutine.

Syntax

A command or function **syntax** is used to show the proper usage and rules that are required to execute it correctly within a script.

System Interface Module

The **System Interface Module** (SIM) is the component of Simphony that allows the System to interface to a variety of other systems, or third-party systems. A special script language known as the ISL provides access to the SIM.

System Variable

A **system variable** is a predefined name that identifies a value which contains information from the Simphony database.

Also see: Argument and Expression

Token

A **token** can be any individual language element inside a script.

Also see: Language Element and Token Error

Token Error

A **token error** can occur any time an individual language element is used incorrectly. For example, incorrect use of whitewashes, missing commas, or erroneous data at the end of a command statement, etc. Please see Appendix A for a complete list of error messages.

Also see: Language Element and Token

TTY

See Asynchronous Serial Interface.

User Variable

A **user variable** is a user-defined name which is assigned a value within a script. The value will remain the same until a newer value is assigned; if no newer value is assigned, the original value is maintained.

Also see: Argument and Expression

Variable

A **variable** is a container whose value changes (the opposite of a constant).

Also see: Constant