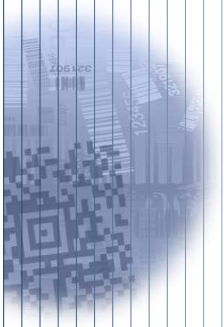


VersaCode™ Language Reference

User's Guide

Disclaimer

Hand Held Products, Inc. d/b/a HHP ("HHP") reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult HHP to determine whether any such changes have been made. The information in this publication does not represent a commitment on the part of HHP.

HHP shall not be liable for technical or editorial errors or omissions contained herein; nor for incidental or consequential damages resulting from the furnishing, performance, or use of this material.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated into another language without the prior written consent of HHP.

© 2000-2002 **Hand Held Products, Inc.** All rights reserved.

Web Address: www.HHP.com

Microsoft® Windows® 95, Windows® 98, Windows® 2000, Windows NT®, Windows® XP, and Windows® Me are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Other product names mentioned in this document may be trademarks or registered trademarks of other companies and are the property of their respective owners.

Table of Contents

Chapter 1 - Getting Started

General Scripting Guidelines 1-1
Sample Scripts 1-1
Start Simply 1-1
Compiler Errors 1-2
Cordless System Implementation 1-2
Script Storage and Use 1-2
Power Management 1-2
RAM Memory Size 1-3
Flash ROM Size 1-3
Background Commands 1-3
IQ App Loader Requirements 1-4
Loading IQ App Loader from CD 1-4
Loading IQ App Loader from the Web 1-5
Using IQ App Loader 1-5
Connecting the Cordless System 1-7
Changing the COM Port 1-8
Serial COM Port Capture 1-8
Compiling a Script 1-9
Compiler Errors 1-10
Customer Support 1-10

Chapter 2 - VersaCode Scripting Language

VersaCode Introduction 2-1
Alphabetic Reference 2-1
VersaCode Programming Example 2-6

User-Defined Variables 2-6
Pre-Defined Variables 2-9
Numeric Expressions 2-9
Boolean Expressions 2-10
String Expressions 2-10
String Range Specifier 2-10
Array Variables 2-11
Type Conversions 2-13
Logical Expressions 2-15
If Statement 2-15
While Statement 2-16
Do While Statement 2-17
Forever Statement 2-18
NOP Statement 2-19
VersaCode Main Programs And Routines 2-21
Introduction 2-21
Program Statement 2-21
Routine Definition Statement 2-21
Execute Statement 2-22
Exit Statement 2-22
A Sample Using Routines 2-23
Locating the Cursor 2-24

Chapter 3 - Programming Models

Introduction 3-1
Scanner Resident 3-2
Host Commands 3-2
Host Routines 3-3
Receiving Input from the Host to a VersaCode Program 3-3
Host Input Send Format 3-3
Host Input Status Return Codes 3-4

Host Command Mode 3-4
Switching to Host Command Mode 3-5
Host Command Send Format 3-5
Host Command Receive Format 3-6
Host Command Error Codes 3-7
Host Command Examples 3-7
Joined Scanners 3-8
Host Routines 3-9

Chapter 4 - Built-In Functions

Introduction 4-1
Conventions 4-1

Chapter 5 - Data Formatting Functions

Introduction 5-1

Chapter 6 - Sample Scripts

No Script (IT3870 Mode) 6-1
History Script 6-5
Part Number and Quantity Script 6-30

Chapter 7 - Glossary

Chapter 8 - Charts

Bitmap Chart 8-1
IT3875 Font Characters Chart 8-4
Numeric Key Codes 8-6

Chapter 9 - Quick Reference Chart

1 Getting Started

VersaCode™ is a script language for building data capture applications that run on HHP products. A VersaCode program can put a prompt on the product's display, get input from the scanner or keypad, format data, send data to the host computer, and signal using the beeper and LEDs.

IQ App Loader™ is a PC tool that allows programmers to edit, compile, and load VersaCode script programs into HHP devices. Another PC tool is available that simplifies the design of applications. With IQ App Designer™ there is no syntax to memorize. Simply point, click, and enter information into forms and IQ App Designer will generate VersaCode script automatically.

This user's guide contains the complete description of the VersaCode language and instructions for the use of IQ App Loader.

General Scripting Guidelines

Before jumping in to write your first VersaCode program, there are some general guidelines to consider. If you have experience with any programming language, such as BASIC, then you will be familiar with the logic and structure of a VersaCode program. A Quick Reference Chart is provided at the back of this user's guide, which lists VersaCode functions according to the type of operation. For example, all of the display functions are listed together. This chart provides the best overview of the functions that are available, and it also serves as a quick reference for syntax.

For more detailed information on each function, refer to the definition in Chapter 4. Examples are provided for each function. In addition, sample VersaCode script programs are provided in Chapter 6. These are full-blown applications, representing the IT3875 factory scripts.

Sample Scripts – Visit versacode.hhp.com

Sample scripts can be used as templates for building custom programs. HHP is introducing a web site where our customers can go to get the latest sample script programs, versacode.hhp.com. There you will find real, working scripts that you can modify to suit your needs.

Start Simply

When designing your first VersaCode program, it is recommended that you start with the simple goal of putting text on the product's display and then build up from there. Not only will you become familiar with the VersaCode functions, but it will also serve to verify that IQ App Loader is correctly installed and that you can communicate with your device. As you add to your program be sure to save periodically and compile periodically. This will make it easier to test each addition to your program.

Compiler Errors

VersaCode programs are compiled to byte code prior to loading into the device. When an error is identified during compilation, the line that contains the error will be identified and a brief description of the problem will be given. Occasionally the compiler will describe a problem that is only a symptom of the root problem. If an error message seems cryptic, use a trial and error process to determine the root problem. For example, the following code snippet will identify the problem as a "missing parenthesis" even though the actual error is that an integer variable is used where a string is expected:

```
String   S1*12   REM declares a string variable of length 12
Int      S2      REM declares an integer variable

REM this produces the compiler error "missing parenthesis"
OutputString(S1 + S2)

REM this line compiles successfully
OutputString(S1 + string(S2))
```

Cordless System Implementation

When designing VersaCode programs for the IT3875 and ST2070 cordless system, there are additional guidelines to follow.

Script Storage and Use

VersaCode programs are stored in the ST2070-5 base unit, along with other work group settings. Each script will be designated to work with one work group. It is possible to store up to four different scripts in the ST2070-5. When a scanner is used to scan the association code on the base, the script for the current work group will be radioed to the scanner and stored in its Flash memory. Once it has been stored in the scanner, it will not be transmitted again until a change is made in the base.

The scanner will need to be configured to run the custom script using the System Setup. Please refer the IT3870/75/5770 System Manual for additional information regarding work groups and scanner settings.

Power Management

When the program is running on a battery-powered scanner, it is important to make use of instructions that allow the scanner to conserve power. When using one of the factory scripts, the IT3875XXX-X2-1 scanner will operate for more than 19 hours assuming typical use of 12 scans per minute, or 720 scans per hour, and a properly conditioned charge pack.

The VersaCode commands InputString and InputNumber both allow the scanner to go to "sleep" while waiting for input, as long as the optional timeout parameter is not used. In each factory script application, the scanner performs a variety of functions and then settles on an input command. This is the typical flow for data capture applications.

Without any power conservation the scanner will draw about 120 mA, which translates into 8 hours of battery life assuming 720 scans per hour.

Use of the display backlight will increase the current draw on the charge pack by about 80 mA when it is illuminated. When using one of the factory scripts with the backlight on continuously, the scanner will operate for about 10 hours between charges.

Note: The WaitForKey function does not allow the scanner to conserve power

RAM Memory Size

RAM is limited to about 20 KB in the IT3875 for running custom scripts. This includes user variables, pre-defined variables, and internal VersaCode interpreter usage, which can vary depending on the number of nested routines. The IT3875 will not run the custom script properly if RAM usage exceeds the available space.

Flash ROM Size

VersaCode script files are stored in the ST2070-5 with a total memory limit of 14 KB, distributed any way between all custom scripts. For example, it is possible to store one script that consumes the entire 14 KB, or four scripts that each consumes 3.5 KB. Script programs are compacted before being stored in the base. The amount of compaction varies, but one can expect aproximately 4X compaction for a VersaCode script and approximately 10X for an IQ App Designer project. For example, a VersaCode script program that fills 8 KB on your hard drive fills about 2 KB of the base's Flash memory. Just before a script is loaded into the base, the PC tool reports the true compacted size of the current script and the space that is available in the base unit.

Background Commands

Most commands must complete execution before the next instruction may begin. For example, the Wait function does not return until the entire time has elapsed.

Some commands operate and execute in the 'background," meaning that the function returns as soon as the operation has been initiated, thus allowing the next VersaCode instruction to begin immediately. For example, the Beep function returns as soon as the beep operation begins. These functions operate in the background:

- Beep can be queued up several commands deep, which means if you provide several Beep commands with or without the same parameters, they will be sounded sequentially with 100 ms silent time between each
- TemporarilyEnableBacklight

- IndicatorBad
- IndicatorGood
- IndicatorLightOn

Note: The indicator functions have a single level deep queue, which means that each operation will terminate as soon as a new indicator command is executed; thus if multiple indicator instructions must be used in sequence, it is recommended to insert Wait instructions between each indicator function

IQ App Loader Requirements

IQ App Loader is designed for use with Microsoft® Windows® 95, 98, Me, NT 4.0, 2000, and XP operating systems. IQ App Loader requires a Pentium-class processor running at 100 MHz, an available COM port, and at least 16 KB of RAM.

In order to store the script in the device using IQ App Loader, the base unit must be connected to the host system with an RS-232 aux port cable and programmed for terminal ID 000. If base is not set up this way, you may use the Visual Menu bar code below to temporarily set up RS-232 and terminal ID 000. After power cycling the base it will revert to the original interface setting. Contact your distributor or HHP representative if you need an Aux Port RS-232 cable for your ST2070 base unit. After the script has been installed, the base may be reconfigured for any interface.

Visual Menu

Loading IQ App Loader from CD

1. Insert the CD labeled IQ App Loader in your CD drive. Follow the screen prompts to install the software if your CD drive uses the AUTORUN.

   OR

   Double click on the **Setup.exe** file. Follow the screen prompts to install the IQ App Loader program.

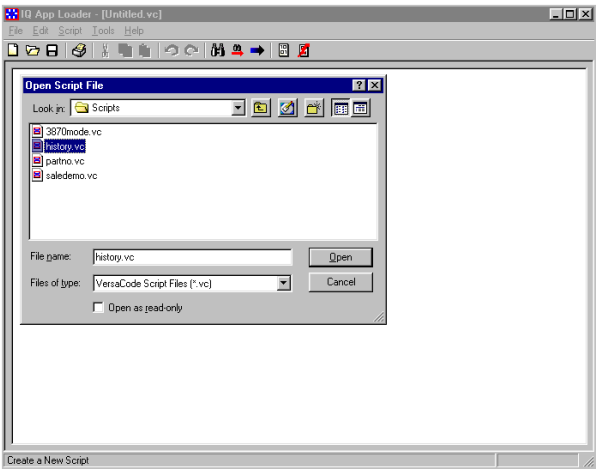
2. To start IQ App Loader, from the Start Menu click on **Programs, HHP**, **IQ App Loader**.

Loading IQ App Loader from the Web

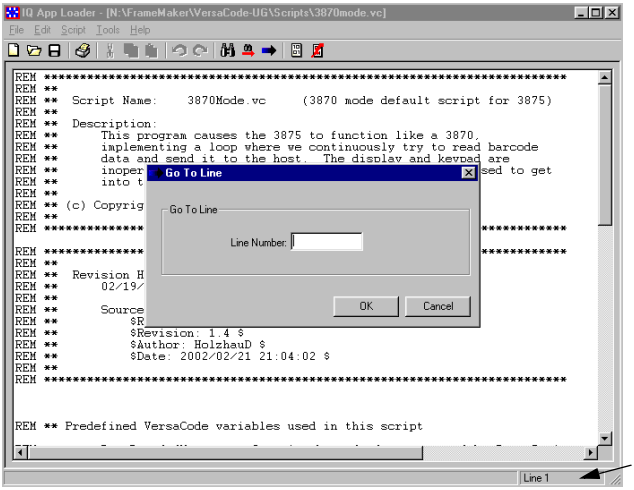
1. Access the HHP web site at www.hhp.com.
2. Click in the **Quick Search** text box and enter IQ App Loader.
3. Click on **Search Now**.
4. Click on the entry for IQ App Loader.
5. When prompted, select **Save File**, and save the files to the **c:\windows\temp** directory.
6. Once you have finished downloading the file, exit the web site.
7. Using Explorer, go to the **c:\windows\temp** file.
8. Double click on the **IQAppLoaderSetup.exe** file. Follow the screen prompts to install the IQ App Loader program.
9. To start IQ App Loader, from the Start Menu click on **Programs, HHP**, **IQ App Loader**.

Using IQ App Loader

You may either write a VersaCode script using your favorite editor, or using IQ App Loader's built-in editor by selecting File - New. If you use another editor, create the file and save it in text format, using ".vc" as the VersaCode file extension. The file can then be opened using IQ App Loader using **File - Open** (or the file open button) to locate and open the script file.

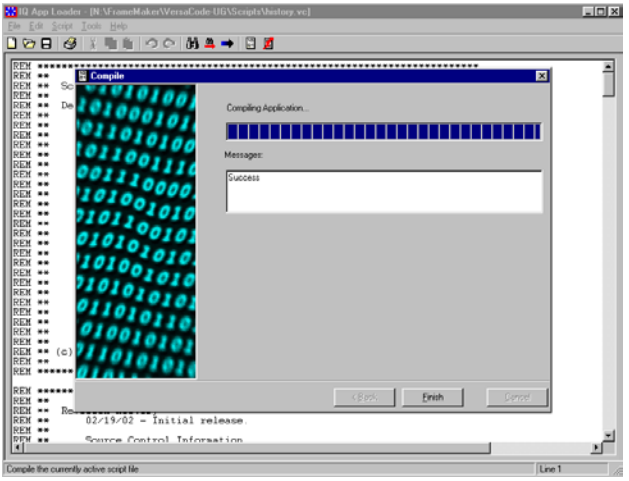


If you want to go to a specific line in your script, you may select the Go To Line option in the **EDIT** menu or the GoToLine icon (right arrow on tool bar).



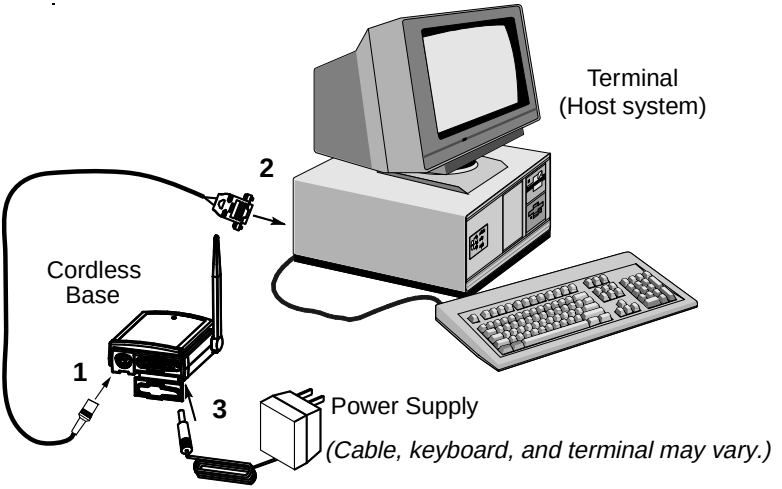
Once you have opened the script file, you can compile it by either clicking on **Script - Compile**, or clicking on the Compile button. This compiles your program and notifies you of any errors.

Note: If your ST2070-5 is connected properly through the COM port of your PC, you can compile and write a script; otherwise, you can only compile a script.



Connecting the Cordless System

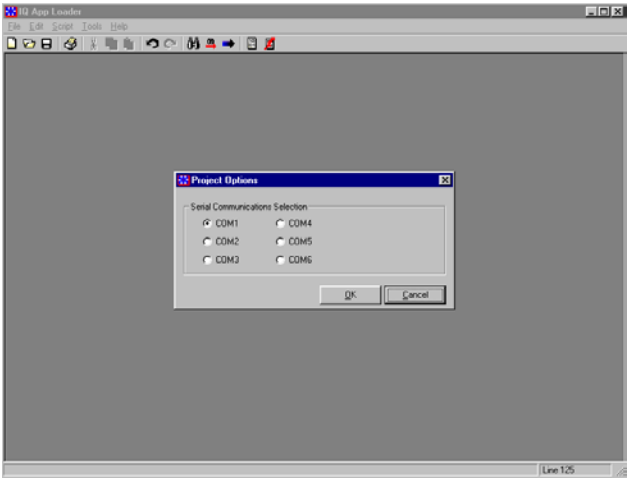
1. Connect your interface cable between the ST2070-5 base unit and the host system (steps 1-3, shown in the illustration below). You also need to use an external power supply. Contact your distributor for more information about ordering power supplies or RS-232 cables.



2. Turn on the power to the host system. Verify that the base is on; the green LED on top of the unit should be on. (The base does not have a beeper.)

Changing the COM Port

Set the COM port setting. Click on **Tools - Options** and select the correct setting.

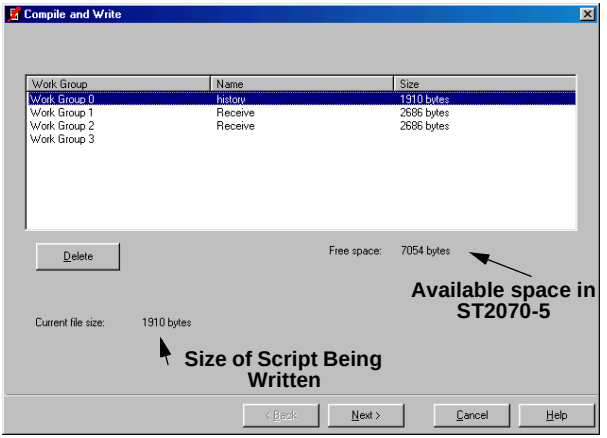


Serial COM Port Capture

IQ App Loader "captures" the serial COM port being used by your PC. Once the script is loaded into the ST2070-5, exit the program to allow other programs to use the COM port.

Compiling a Script

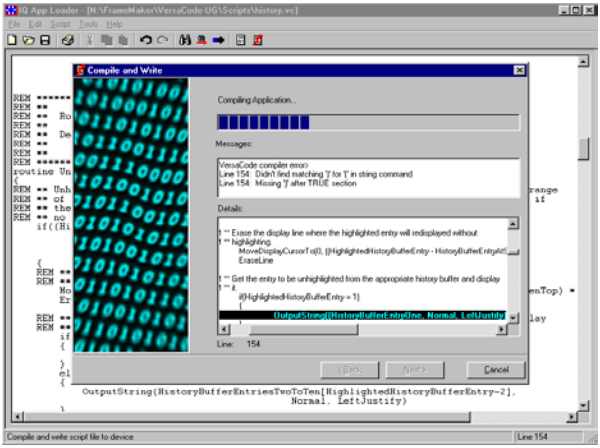
When you are working online and are ready to write the program to the base unit, click on **Script - Compile and Write**, or click on the Compile and Write button. Select the work group into which you want the script loaded. (Refer to the Cordless System Manual for additional information about work groups.) If you select a work group that already contains a script, you will be asked if you want to overwrite the current script. Your program is then downloaded to the base unit, if there are no errors. If there are any errors, the program is not downloaded, and an explanation appears in the Compile Script window. The screen below shows that a script called History (1910 bytes) is in work group 0.



Note: If there is not enough memory in the base unit to write your script, you will have to delete an existing script first. Select the script to delete and select the **Delete** button.

Compiler Errors

The compiler quits when there is a syntax error. A message appears describing where the first failure occurred and why. To correct, consult the referenced command and check the syntax.



Customer Support

If you need assistance using VersaCode or IQ App Loader, please contact your Distributor or the nearest HHP technical support office:

North America:

Telephone:(315) 685-2476 (8 a.m. to 8 p.m. EST)
or, in the U.S.: (800) 782-4263
Fax number:(315) 685-4960
E-mail: tech_support@hhp.com

Europe:

Telephone: European Ofc: +31 (0) 40 29 01 600
U.K. Ofc: +44 1925 240055
E-mail: euro_support@hhp.com

Asia:
Telephone:+852-2511-3050 *or* 2511-3132
E-mail: asia_support@hhp.com

Latin America:
Telephone:+(56) 2 942-2495 *or* (56) 2 942-8371
E-mail: la_support@hhp.com

2 VersaCode Scripting Language

VersaCode Introduction

The VersaCode language is most similar to BASIC, with built-in support for data capture applications.

The language includes functions for performing such operations as: display text or images, get input from scanner, keypad, or host computer, comparisons, send data to host, and signal using the beeper and LEDs.

Alphabetic Reference

Name	Description	Page
BatteryStrength	Returns battery strength	4-2
Beep	Sounds beeper	4-2
CaseInsensitiveCompare	Compares 2 strings in case insensitive manner	4-3
CaseInsensitiveWildcardCompare	Compares 2 strings, using wildcard * and ?, in case insensitive manner	4-4
CaseSensitiveCompare	Compares 2 strings in case sensitive manner	4-6
CaseSensitiveWildcardCompare	Compares 2 strings, using wildcards * and ?, in case sensitive manner	4-7
CharacterHeight	Gets height of characters (in pixels)	4-8
CharacterWidth	Gets width of characters (in pixels)	4-9
CheckForCommand	Checks to see if host VersaCode command has been received	4-11
Comment	Allows comments to be added to code (see also REM, page 2-9)	2-9
CompareNextCharacterTo	Compares character at EditCursor to x	5-1
DefaultPrefix	Sets default prefix used by Send function	4-12
DefaultSuffix	Sets default prefix used by Send function	4-12
DisplayBacklightOff	Turns off backlight	4-13

Name	Description	Page
DisplayBacklightOn	Turns on backlight	4-14
DisplayCursorOff	Turns display cursor off	4-15
DisplayCursorOnBlink	Makes display cursor blink	4-15
DisplayCursorOnSteady	Turns display cursor on steadily	4-16
DisplayHeight	Gets height of output display (in pixels)	4-16
DisplayLanguage	Returns a value for selected display language	4-17
DisplayMenu	Constructs text menu on display	4-17
DisplayWidth	Gets width of output display (in pixels)	4-20
Do While Statement	Program loop control	2-17
EraseDisplay	Erases display	4-20
EraseLine	Erases line(s) in display	4-21
Execute Statement	Passes control to a routine	2-22
Exit Statement	Terminates a routine	2-22
Expressions		
Boolean Expressions	Expressions assigned to Boolean variables	2-10
Logical Expressions	Returns a Boolean value based on the outcome of a logical test	2-15
Numeric Expressions	Expressions assigned to numeric variables or used as arguments to functions that expect a number	2-9
String Expressions	Expressions assigned to string variables or used as arguments to functions that expect a string	2-10
Find	Finds string 1 in string 2	4-22
Forever Statement	Program loop control	2-18
GetFont	Gets index value of active font	4-24
GetKey	Gets scan code of oldest keypress in keypad buffer	4-25
GetTimer0	Returns value of system timer (in milliseconds)	4-27
GoTo Statement	Program control	2-19

Name	Description	Page
If Statement	Program Selection Control	2-15
IndicatorBad	Initiates the bad LED/beeper sequence	4-28
IndicatorGood	Initiates the good LED/beeper sequence	4-29
IndicatorLightOff	Turns off all indicator lights	4-29
IndicatorLightOn	Cycles an indicator light on and off	4-30
InputNumber	Waits for number from device	4-31
InputString	Waits for a string from device	4-36
InterpreterVersion	Displays interpreter version	4-40
IsNextCharacterNonNumeric	Determines if character at EditCursor is numeric	5-1
IsNextCharacterNumeric	Determines if character at EditCursor is numeric	5-2
KeyFlush	Turns off audible keypad button clicks	4-40
KeyLook	Gets scan code of oldest keypress in keypad buffer	4-41
LeftJustify	Left justifies a string	4-43
Label	Label name	2-19
Length	Gets number of characters in string	4-44
LoadImage	Loads image to display	4-45
LowerCase	Converts alpha characters to lower case	4-47
MenuCommand	Process string as menu command	4-47
MoveCursorAheadBy	Moves cursor ahead by n on data	5-2
MoveCursorAheadToChar	Moves cursor ahead to character x on data	5-2
MoveCursorAheadToNonChar	Moves cursor ahead to character that is not x on data	5-3
MoveCursorBackBy	Moves EditCursor back by n on data	5-3
MoveCursorBackToChar	Moves cursor back to character x on data	5-4
MoveCursorBackToNonChar	Moves cursor back to character that is not x on data	5-4

Name	Description	Page
MoveCursorToBeginning	Moves cursor to beginning of edited data	5-4
MoveCursorToEnd	Moves cursor to end of edited data	5-5
MoveDisplayCursorTo	Moves cursor to specific location on display	4-48
NOP Statement	No function	2-19
OutputString	Writes string to the display	4-49
Program Statement	Defines start of main program	2-21
RadioStrength	Returns radio signal strength	4-51
REM	Allows comments to be added to code (see also Comment, page 2-9)	2-9
ReplaceCharacters	Specifies replacement characters in edited data	5-5
ResetTimer0	Resets system timer to zero	4-51
RightJustify	Right justifies a string	4-52
Routine Definition Statement	Defines start of routines	2-21
ScannerID	Returns scanner ID	4-53
ScrollDisplay	Scrolls display up or down	4-54
Send	Sends string to host port	4-55
SetFont	Sets active font for display	4-49
SetLength	Sets length for string variable	4-58
StringComparisonsAreCaseInsensitiveByDefault	Default string comparisons to case insensitive	4-59
StringComparisonsAreCaseSensitiveByDefault	Default string comparisons to case sensitive	4-60
SuppressCharacters	Specifies characters to be suppressed in edited data	5-5
TemporarilyEnableBacklight	Temporarily backlights display for specified time	4-61
TriggerHit	Checks for trigger press	4-62
TurnOffCharacterReplacement	Turns off character replacement in edited data	5-6
TurnOffCharacterSuppression	Turns off character suppression in edited data	5-6

Name	Description	Page
Type Conversions		
Boolean	Allowed type conversion	2-13
CharToInt	Allowed type conversion	2-15
Int	Allowed type conversion	2-13
IntToChar	Allowed type conversion	2-15
String	Allowed type conversion	2-14
UpperCase	Converts alpha characters to upper case in data	4-62
Variables		
Array Variables	Describes array variables	2-11
Pre-Defined Variables	Preserved variables	2-9
User-Defined Variables	User variables: Int, String, Boolean	2-6
Do While Statement	Program loop control	2-17
Wait	Waits for specified time	4-63
WaitForCommand	Waits for host VersaCode command from host port	4-64
WaitForKey	Gets scan code of oldest keypress in keypad buffer and removes it from buffer	4-65
WaitForMultipleCommands	Waits for multiple host VersaCode commands from host port	4-67
WildcardCompare	Compares 2 strings, using wildcards * and ?	4-68
Workgroup	Returns current work group number	4-70
WriteAllButLastCharactersStartingAtCursor	Sends characters from EditCursor to end, less n on data	5-6
WriteCharactersFromCursor	Send next n characters from EditCursor on data	5-7
WriteCharactersFromCursorToChar	Sends characters from EditCursor to first character x on data	5-7
WriteCharactersFromCursorToEnd	Sends characters from EditCursor to end on data	5-8
WriteCharMultipleTimes	Send character x, n times on data	5-8

VersaCode Programming Example

When programming in VersaCode you use built-in functions, such as OutputString, to perform operations. You may also define and use your own variables for storing data. All keywords and variable names are case insensitive. In other words, OutputString, outputStRiNg, OUTPUTSTRING, and outputstring are all interpreted the same way.

Here is a simple example program to read data from a keypad, then output the data with a text description to both a serial port and a display:

```
Comment Sample Program
string StringData*32

InputString(StringData)
StringData = "Input Data = " + StringData
OutputString(StringData)
Send(StringData+"…")
```

Line 1 Comment indicates that the remainder of the text on that line is not to be treated as actual program data, but is there for documentation purposes.

Line 2 String indicates that one or more string variables are to be declared. StringData is the name of the string variable being declared, and the *32 indicates that StringData stores up to 32 characters.

Line 4 InputString attempts to read data from either the scanner or the keypad, and stores the data into the variable StringData.

Line 5 Input Data = and the original contents of StringData are concatenated together by the + operand and the result is stored back into StringData.

Line 6 *OutputString* causes the text stored in StringData to be written to the scanner display.

Line 7 *Send* causes the text stored in StringData, with the text "…" concatenated to the end of StringData, to be written to the scanner host port.

Variables are explained in detail in User-Defined Variables, page 2-6. The functions InputString, OutputString, and Send are explained in *Built-In Functions* beginning on page 4-1.

User-Defined Variables

It is possible to define storage variables to which data may be assigned. There are three types of storage variables: numeric, Boolean, and string. You can convert from one type of variable to another. See "Type Conversions" on page 2-13.

These general guidelines should be followed when choosing variable names:

- The variable name must consist of a minimum of 1 and a maximum of 255 characters.
- The variable name must start with an alphabetic character. Each remaining character must be either alphanumeric or an underscore.
- Variable names must not conflict with either a VersaCode keyword or a previously declared name or label.

Numeric variables are used to store integer numbers in the range –2147483648 to 2147483647. The following is an example of numeric variables and the operations that can be performed on them:

```
Int    N1, N2, N3
Int    N4
String S1*32

N1=3*5
N2=8
N3=N1/N2
N4=N3-N2
N1=N1+InputString (S1)
```

In this sample, four numeric variables (N1, N2, N3, and N4) are declared. String S1 was declared to be used in the InputString function.

Line 4 Leaves N1 with the value 15

Line 5 Leaves N2 with the value 8

Line 6 Leaves N3 with the value 1

Line 7 Leaves N4 with the value –7

Line 8 Gets scan or keypad input, stores it to S1 (which we're not using, but is a required parameter to InputString), and returns the number of characters read. This number is added to N1, with the result being stored back to N1.

Boolean variables have a value of either true or false, and are used to indicate whether or not a condition was met. The following is an example of Boolean variables and the operations that can be performed on them:

```
Int   N1,N2
Boolean B1,B2

N1 = 3
N2 = 4
B1 = (N2 > N1)
B2 = B1 AND (N2 <> N1)

If(B2)
{
      Send("N2 is greater than N1, N2 is not
equal to N1")
}
else
{
      Send("Either N2 is no greater than N1, or
N2 is equal to N1")
}
```

In this example, two Boolean variables are declared: B1 and B2. N1 and N2 are declared as numbers with values 3 and 4, respectively. In the B1 assignment statement, a logical operation is performed to see if N2 is greater than N1; B1 is set to true if it this is the case, and FALSE otherwise. In the B2 assignment statement, B2 is set to true if both B1 is true and N2 is not equal to N1. (See Logical Expressions on page 2-15 for more details.) A test is then run via the If statement to find if B2 is true. The first send statement is executed if B2 is true, and the second is executed if B2 is false. See If Statement on page 2-15 for more details.

String variables are used to store text. The declaration is similar to that of numbers and Booleans, but the maximum number of characters must also be specified. The form of the declaration is to use the keyword String followed by variables, separated by commas. Each variable name in the declaration must be immediately followed by an asterisk and a numeric constant that specifies its maximum size. The following is an example of string variables and the operations that can be performed on them:

```
String S1*48,S2*64
S1 = "1234567890abcdefghijklmnopqrstuvwxyz"
S2 = S1
S2 = S2 + S1
```

In this example, two string variables are declared: S1 and S2. S1 is assigned some text and then S1 is copied to S2. Line 4 does not work, because concatenating S1, which is 36 characters in length, to S2, which is also 36 characters in length, exceeds the maximum of 64 characters allowed in S2.

Pre-Defined Variables

There are several keywords in VersaCode that behave as variables:

LastDecodedMessage is a pre-defined string that contains the last data read from a bar code scan. It may be used in the same manner as a string variable. If you assign a string value to it, you overwrite the symbol information that was stored in it. See *Data Formatting Functions* beginning on page 5-1 for further information.

EditCursor is a keyword used by data formatting functions described in Data Formatting Functions (see page 5-1). It is used to mark a position in LastDecodedMessage. It may be used in the same manner as a numeric variable.

Host is a keyword that may be used in the same manner as a string variable. If you assign a string to it, the string is written out the host port of the scanner. If you assign Host to a string, any data waiting to be read in the host port buffer is written to the assignment string.

Comment
REM

To document the operations performed in a VersaCode program, use either the keyword "Comment" or the Basic language keyword "REM" to denote that all text on the same line after the keyword should be ignored by VersaCode.

Numeric Expressions

Numeric expressions may be assigned to numeric variables or used as arguments to functions that expect a number. Numeric expressions may consist of any combination of numeric constants, numeric variables, and functions that return a numeric value, separated by numeric operators. The numeric operators are '+' for adding, '-' for subtracting, '*' for multiplying, '/' for dividing, '&' for bitwise AND, '|' for bitwise OR, and '^' for bitwise XOR. In addition, a unary '~' before a numeric expression causes the expression to be ONEs complemented. When evaluating numeric expressions, the precedence is as follows:

- Unary +, unary -, and unary ~ are highest priority. These are performed on the corresponding operand, then used in the numeric expression.
- * and / are second highest priority. These are evaluated before any operators at a lower priority. Evaluation of more than one these operators is in left to right order in the expression.
- +, -, &, |, and ^ are lowest priority. Evaluation of more than one of these operators is in left to right order in the expression.

Placing part of a numeric expression with parenthesis forces that part to be evaluated first before the result is evaluated along with the remainder of the numeric expression.

Example: If a, b, c, d, e, f, and g are integers and a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, and g = 7, then a + b - c * d / e - f * -g returns 43. Likewise, a + (b - c) * d / (e - f) * -g returns -27.

Boolean Expressions

Boolean expressions may be assigned to Boolean variables. Boolean expressions used as arguments to functions that expect a Boolean value, or used in conditional statements (see If Statement, page 2-15, While Statement, page 2-16, and Do While Statement, page 2-17), may consist of any combination of Boolean constants (keywords TRUE or FALSE), Boolean variables, and functions that return a Boolean value, separated by Boolean operators. The Boolean operators are 'AND' for logically ANDing two operands, and 'OR' for logically ORing two operands.

String Expressions

String expressions may be assigned to string variables or used as arguments to functions that expect a string. String expressions may consist of any combination of string constants enclosed in single or double quotes, string variables, and functions that return a string value, separated by string operators. The only string operator available is '+' for concatenating two strings values together. String constants may be enclosed in either single or double quotes when you want to use a single or double quote character in a constant. If you want to create the text string "Doesn't this work?", you could use the following string expression: '"Doesn' + "'" + 't this work?"'.

If you are searching for a question mark (?) or asterisk (*) character in a string, enter the search beginning with a caret (^). For example, to search for "A?B*C," enter the search string "A^?B^*C." If you are searching for a caret mark, use two marks (^^). For example, to search for "A^B," enter the search string "A^^B."

The maximum allowable size of a string variable is 3000 bytes. This accommodates the largest PDF417 symbol that may be read by an IT3875.

String Range Specifier

It is possible to select a range of characters within a string and use it as part of a string expression. The range is specified after the string variable with parenthesis surrounding two numeric expression, separated by commas. The first numeric expression specifies the start character and the second expression specifies the end character. Both numeric expressions must evaluate to a number that is greater than or equal to zero (the first position in the string) and is

less than the number of characters in the string. If the first numeric expression is greater than the second numeric expression, then the range of characters is reversed. The following sample illustrates usage of the range specifier. The first OutputString statement outputs “cd” and the second outputs “fedcba”.

```
String S1*32

S1="abcdefg"
OutputString(S1(2,3))
OutputString(S1(5,0))
```

It is also possible to assign a value to a range of character positions in a string variable by using a range expression, as defined above, on the string variable that is to be written to. This capability allows individual characters to be altered in a string variable without having to rewrite the entire contents of the variable. For example, you might wish to examine every character of the string for the letter k, and change it to the letter j. There are several aspects of this method that are not obvious:

- The numeric expressions for the beginning and end of the range to be written must be less than the maximum declared length of the string variable.
- The string length is set automatically during a normal assignment. For example, S1=”abcde” sets the value of string variable S1 to “abcde” with a length of 5 characters. If you then enter S1(6,7)=”gh” to assign character ‘g’ to the 7th position and ‘h’ to the 8th position, the length is reset to 8 and the 6th position contains an undetermined value. If you wish to set the length to a fixed value, see SetLength, page 4-58.
- If the character range in a string variable is too small for the number of characters in the string expression being assigned to it, characters are truncated from the right of the string expression in order to fit within the range. If, for example, you assign S1(4,6) = “qrstuv”, character ‘q’ is written to the 5th position, ‘r’ to the 6th position, ‘s’ to the 7th position, and characters ‘t’, ‘u’, and ‘v’ are lost.
- If there are not enough characters in the string expression to fill the character range in a string variable, then the characters that are available are written starting at the beginning of the specified range until there are no more characters to write. If, for example, you assign S1(4,7) = “jk”, ‘j’ is written to the 5th position, ‘k’ is written to the 6th position, and the 7th and 8th positions remains unchanged.

The maximum allowable size of a string variable is 3000 bytes. This accommodates the largest PDF417 symbol that may be read by an IT3875.

Array Variables

It is possible for a single variable to consist of many data elements by making it multi-dimensional. A variable that is declared to be multi-dimensional is called an array variable. Array variables may consist of up to five dimensions. When an array variable is declared, the maximum size of each dimension is enclosed in square brackets following the name of the variable being declared. In the case of a string declaration, the bracketed dimensional information comes after the maximum size of each string element. When an array variable element is referenced, the variable name is followed by a square bracketed numeric

expression for each dimensional element declared for that variable. The numeric expressions are used to pinpoint the exact data element to be referenced. The following is an example of an integer array variable implementation. In this example, numbers are read in order to find the first occurrence of the number 10:

```
Int   I1[5][6]
Int   Row,Column

Row=0
While(Row < 5)
{
      Column = 0
      While(Column < 6)
      {
      InputNumber(I1[Row][Column])
      Column = Column + 1
      }
      Row=Row+1
}

Row=0
Column = 0
While((Row < 5) AND (I1[Row][Column] <> 10))
{
      Column = Column + 1
      If(Column >= 6)
      {
      Column = 0
      Row = Row + 1
      }
}

if(Row < 5)
{
      OutputString("10 found at
"+string(Row)+","+string(Column))
}
else
{
      OutputString("No 10 found!")
}
```

This example first declares an integer array I1 which contains 5 rows and 6 columns of data for a total of 30 data elements. The user must input 30 numbers, each of which is put into a unique element of the array. A search is performed for the first occurrence of the number 10. If it is found, the row and column where it was found is displayed.

Here is an example using a string array:

```
String S1*40[6]
Int   I1

S1[0] = "Joe"
S1[1] = "Kathy"
S1[2] = "Karen"
S1[3] = "Dave"
S1[4] = "Mike"
S1[5] = "Rob"

If(InputNumber(I1, Minimum=0, Maximum=5))
{
      OutputString("Hi "+S1[I1])
}
```

In this example, a one dimensional string array S1 consisting of 6 elements is declared. This string array is loaded with names. The user is asked for a number in the range of 0 to 5 and, if it is received, a greeting is printed using the input number as an index into the name table.

Type Conversions

Type conversions convert data from one type to another. For example, when trying to see the value of a variable, you may want to display it in text (or String) form. You may also have a string containing a number that you want to convert to an integer variable. The following is a list of conversion operations:

Int(Boolean expression) converts the Boolean expression to an integer if the expression evaluates to true, and to a numeric zero if the expression evaluates to false. The following sample program leaves a value of 0 in J and 1 in K:

```
Int   j,k
Boolean b

B = false
J = B
K = NOT B
```

Int(string expression[, integer variable]) searches for a number at the beginning of the string expression and if found, converts it to an integer. Spaces and tabs may precede the number. The search for a number stops when the end of the string expression is reached or when a non-numeric character is

encountered. Optionally, an integer variable may be included as the second parameter, which contains the number of characters used in the conversion of the string expression to a number. The following sample program leaves a value of 123 in J and K and a value of 3 in M:

```
Int   j,k,m
String S*32

S="   123abcd"
J=int(S)
K=int(S,M)
```

String(Boolean expression) converts the Boolean expression to the constant string “TRUE” if the expression evaluates to true and to “FALSE” if the expression evaluates to false. The following sample program leaves a string value of “FALSE” in S1 and “TRUE” in S2:

```
stringS1*32,S2*32
Booleanb

B = false
S1 = B
S2 = NOT B
```

String(numeric expression) converts the numeric expression to a string with no fill characters. The following sample program leaves a string value of “123” in S1, “-123” in S2, and “0” in S3:

```
String S1*32, S2*32, S3*32
Int   j

J=123
S1=string(J)
S2=string(-J)
S3=string(J-J)
```

Boolean(string expression) returns a value of false if the string is null and true otherwise. The following sample program leaves a value of false in B1 and true in B2:

```
Boolean B1,B2
StringS*32

S="   123abcd"
B1=""
B2=Boolean(S1)
```

Boolean(numeric expression) returns a value of false if the numeric expression evaluates to zero and true otherwise. The following sample program leaves a value of false in B1 and true in B2:

```
Boolean B1,B2
Int   j

J=123
B1=string(J-J)
B2=string(-J)
```

CharToInt(string expression) returns the ASCII value of the first character in the string expression. The following sample program leaves a value of 65 in J:

```
Int   j

J = chartoint('A')
```

IntToChar(numeric expression) returns the character corresponding to the ASCII value specified by the numeric expression. The following sample program leaves a string value of "1" in S1:

```
String S1*32

S1=inttochar(49)
```

Logical Expressions

Logical expressions, which may be used to test a number against a number, or a string against a string, return a Boolean value based on the outcome of the test. The syntax is (V1 operator V2), where the left and right parenthesis are required, V1 and V2 are either numeric or string expressions, and the operator may be "<", ">", "<=", ">=", "=", and "<>" (not equal). For numbers, these comparison operators are straightforward. For string expressions, the "=" (equal) and "<>" (not equal) operators are straightforward. The less than and greater than operations evaluate in the following manner: one string is considered less than a second string if the first non-equal character is less than the corresponding character in the second string or if the strings are equal up to the end of the first string, but the second string is longer than the first string.

If Statement

The If statement is used to conditionally perform VersaCode operations depending upon whether or not a Boolean expression evaluates to true or false. The syntax is of the following form:

```
IF(Boolean expression) [THEN] (statements)
[[ELSE] (statements)]
```

The Boolean expression must always be enclosed in parentheses. The first set of VersaCode program statements, which may be enclosed in either parentheses or curly brackets, must always be present and may optionally be

preceded by the word THEN. These statements are executed if the Boolean expression evaluates to true. The second set of VersaCode program statements are optional and may be enclosed in either parentheses or curly brackets and may optionally be preceded by the word ELSE. These statements are executed if the Boolean expression evaluates to false.

The following sample program causes the text “J is greater than or equal to K” and “B1 is true” to be sent to the display:

```
Boolean B1,B2
Int   j,k

J=123
K=24
B1=true
B2=(j<k)
If(B2)
{
      outputstring("J is less than K")
}
else
{
      outputstring("J is greater than or equal
to K")
}
If(B1)
{
      outputstring("B1 is true")
}
else
{
      outputstring("B1 is false")
}
```

While Statement

The While statement is used to repeatedly perform a set of VersaCode operations as long as a Boolean expression evaluates to true. The syntax is of the following form:

```
WHILE(Boolean expression) [DO] (statements)
```

The Boolean expression must always be enclosed in parentheses. The set of VersaCode program statements, which may be enclosed in either parentheses or curly brackets, must always be present and may optionally be preceded by the word DO. The Boolean expression is evaluated and the statements are executed repeatedly as long as the Boolean expression evaluates to true. If the Boolean expression evaluates to false on the first iteration, the statements are never executed.

In the following sample program, the statements in the first While are not executed, since the Boolean expression evaluates to false immediately. The statements in the second While are executed 99 times (until k is incremented to 123), and the value of k is output at each iteration.

```
Int   j,k

J=123
K=24
While(j<k)
{
      outputstring("J = "+string(j))
      j = j + 1
}
While(k<j)
{
      outputstring("K = "+string(k))
      k = k + 1
}
```

Do While Statement

The Do While statement is used to repeatedly perform a set of VersaCode operations as long as a Boolean expression evaluates to true. The syntax is of the following form:

```
DO (statements) WHILE(Boolean expression)
```

The Boolean expression must always be enclosed in parentheses. The set of VersaCode program statements, which may be enclosed in either parentheses or curly brackets, must always be present and may optionally be preceded by the word DO. The Boolean expression is evaluated and the statements are executed repeatedly as long as the Boolean expression evaluates to true. If the Boolean expression evaluates to false on the first iteration, the statements have already been executed once. Unlike the While statement, the Do While executes the statements first and then evaluates the Boolean expression.

In the following sample program, the statements in the first Do While is executed once, causing the string “J = 123” to be sent to the display and J to be incremented to 124. The second Do While is executed 100 times (until k is incremented to 124), and the value of k is output at each iteration.

```
Int   j,k

J=123
K=24
do
{
      outputstring("J = "+string(j))
      j = j + 1
}
While(j<k)
do
{
      outputstring("K = "+string(k))
      k = k + 1
}
While(k<j)
```

Forever Statement

The Forever statement is used to repeatedly perform a set of VersaCode operations. To move forward in the program from this statement, use the Execute Statement (page 2-22), Exit, or GoTo. The syntax is of the following form:

```
FOREVER (statements)
```

The set of VersaCode program statements may be enclosed in either parentheses or curly brackets and may optionally be preceded by the word DO.

In the following sample program, the statements are executed until the user inputs “123” via the keypad or bar code scanner.

```
String S1
Int j, k

Forever
{
      InputString(S1)
      If(S1 = "123")
      {
      Exit
      }
}
j=123
k=24
While(j<k)
{
      outputstring("j = "+string(j))
      j = j + 1
}
While(k<j)
{
      outputstring("k = "+string(k))
      k = k + 1
}
```

NOP Statement

A NOP statement is provided that performs no function other than to take up execution time.

GoTo Statement
Label

The GoTo statement transfers control immediately to a statement label in a program. The syntax of a statement label is:

name:

where name is the name of the statement label. The syntax of a GoTo statement is:

GoTo name

where name is the name of the statement label. The following rules apply:

- A statement label may be defined within any routine or within the main program, but may not be defined within an IF, WHILE, DOWHILE, or FOREVER statement.
- A statement label name must comply with the same naming rules as User Defined Variables.
- The name of a statement label must be unique within the routine or main program in which it appears.
- A GoTo can only reference statement labels defined in the same routine or main program in which the GoTo appears.

In the following sample program, "12345FinalOutput" is written to the display. The GoTo takes place during the fifth iteration of the Forever loop, and skips the "IntermediateOutput" display instruction.

```
Int     I1

 I1 = 1
 Forever
 {
       OutputString(String(I1))
       I1 = I1 + 1
       if(I1 > 5)
       {
       GoTo FinalOutput
       }
 }

 OutputString("IntermediateOutput")

FinalOutput:
 OutputString("FinalOutput")
```

VersaCode Main Programs And Routines

Introduction

As programs become complex, it is often useful to break them up into routines that perform specific functions. VersaCode provides a routine definition (Routine statement) and execution capability (Execute statement), and the means for identifying the main program via the Program statement. All variable declarations are global in scope, including those that are declared inside a routine. Group variables that will be global in scope outside routines for compatibility with future compiler releases. Routines can only take one optional string argument. The value of the string expression is written into LastDecodedMessage for use by the routine being called; the value of LastDecodedMessage reverts to its former value after control is returned from the routine. The Execute Statement (page 2-22) may be used at any point in a routine to return control to its caller. Exit may be used in the main program to terminate the VersaCode program execution.

Program Statement

The Program statement defines the start of the main program. When you are breaking a program up into routines, you need to explicitly mark the main program statements to separate them from routines. Put the word "Program" at the beginning of the main body of statements and the word "End" at the end of the statements. When the VersaCode program starts executing, control is passed to the first executable statement after Program. If the word Program isn't present, then routines should not be used. If no Exit statement is encountered beforehand, the program statements are executed sequentially, terminating at the last statement before the End keyword. The following sample demonstrates the syntax:

```
Program
OutputString("This is a sample main program")
End
```

Routine Definition Statement

The Routine statement defines the start of a new routine and provides a name for the routine that other routines and the main program use. At the end of the statements in the routine, the word "End" should be included to mark the end of the routine. If no Exit statement is encountered beforehand, then the routine statements are executed sequentially, terminating at the last statement before the End keyword and returning control to the caller. The following sample demonstrates the syntax:

```
Routine Demo
OutputString("This is a demonstration routine")
End
```

Execute Statement

The Execute statement is used to pass control to a routine and to optionally set the value of LastDecodedMessage for the duration of the execution of that routine. The following sample demonstrates the syntax for the Demo routine, where the value “abcd” is loaded into LastDecodedMessage:

```
Execute(Demo, "abcd")
```

Exit Statement

The Exit statement allows premature termination of a routine (returning control to the caller), or termination of the main program. The following sample program terminates prematurely if the user enters the word Exit:

```
string StringData*32

OutputString("Input a string: ")
InputString(StringData)
If(StringData = "Exit")
{
      Exit
}

OutputString("You didn’t tell me to exit!")
```

A Sample Using Routines

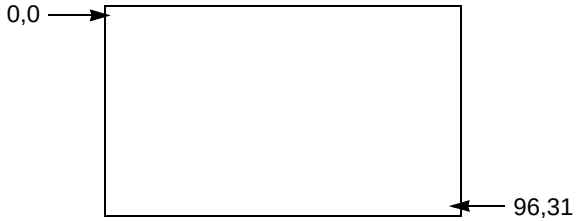
The following sample demonstrates the use of routines in a program:

```
String S1*100

Routine GetInput

Do
{
      InputString(S1)

      If(S1 = "Exit")
      {
      Exit
      }
} while (S1(0,1)<>"AB")

End

Routine DisplayOutput

OutputString("Input = " + LastDecodedMessage)

End

Program

Execute(GetInput)
Execute(DisplayOutput, S1)

End
```

The main program calls GetInput, which continually tries to get input from the user into the global variable S1. If the user enters the word “Exit”, GetInput quits. Otherwise, GetInput keeps trying to get input until the first two characters in the user input are “AB”. When GetInput returns control to the main program, the main program calls DisplayOutput with S1 as an argument, which is written into DisplayOutput’s copy of LastDecodedMessage. DisplayOutput writes the message to the display and returns.

Locating the Cursor

The format for an HHP device display is x,y, with 0,0 indicating the upper left corner, and 96,31 the lower right.



Note: When locating a cursor or an image, the reference coordinates used are at the top left.

3 Programming Models

Introduction

The VersaCode program may be designed using one of three different programming models, each with its own benefits. Please refer to the table below for a summary of each model. Each model is suited for multiple scanner environments and each brings the full power of VersaCode to your data capture applications.

Programming Model	***Description***	***Benefit***	***Interface***
Scanner Resident	VersaCode programs stored in base and run in the IT3875XXX-X2-1	No need to modify host software Works with any interface	Any interface (RS-232 is required to load script into base during setup, but any interface may be used at run time)
Host Commands	VersaCode commands sent from host to the scanner through base	Host maintains complete control over scanner Priority on system updates: never any changes to scanner or base	Aux port RS-232 is required
Host Routines	VersaCode routines are resident in scanner and activated using Host Commands	Host computer maintains complete control over scanner Priority on system performance	Aux port RS-232 is required

Scanner Resident

Scanner Resident is the most common programming model. The VersaCode program is compiled and loaded into the device, and runs autonomously in the device. No host programming is required. The host may or may not send input to the device during operation. If host input is required it is accomplished using the InputString (or InputNumber) function in the VersaCode program, and the host sends input according to the format described in Host Input commands (see Receiving Input from the Host to a VersaCode Program on page 3-3). If no input is required from the host during operation then it is possible to use any interface between host and device during operation, including one-way interfaces such as keyboard wedge.

Host Commands

Using this programming model the host computer has complete control over the device. VersaCode commands are sent from the host to the device during operation according to the format described in Host Commands (see Host Command Send Format on page 3-5). Since the device expects the commands from the host, it is necessary to load a simple program into the device. That program may be as simple as the one below, which instructs the device to do nothing but wait for commands to be sent by the host:

```
Program
{
    WaitForMultipleCommands
}
End
```

Since the program contains no specific instructions, it is possible to construct the program entirely in the host. This may be important for large-scale deployments where it is anticipated that program updates will be made on a periodic basis. Since all the intelligence is in the host, it is only necessary to push host software upgrades to every site, and no device changes are ever needed. The ability to receive commands from the host computer does require a two-way interface, such as RS-232.

Host Routines

VersaCode is the first script language in the industry to support a hybrid programming model. In the hybrid model, a VersaCode program is resident in the device, and at specified points during operation it instructs the device to await a command from the host computer using one of the instructions: CheckForCommand, WaitForCommand, or WaitForMultipleCommands. The host may send any Host Command, or it may instruct the device to execute a VersaCode routine according to the format described in Host Routines (see Host Routines on page 3-9). Those routines must be designated as Host Routines in the VersaCode program that is resident in the device.

Often the scanner must perform numerous VersaCode instructions. It is faster to group those instructions into a routine and to have the host call that routine, than to have the host send each instruction as an individual Host Command. This is the way to achieve the highest system performance while the host computer maintains control of the device. The ability to receive commands from the host computer requires a two-way interface, such as RS-232.

Receiving Input from the Host to a VersaCode Program

Note: This section is appropriate for Scanner Resident and Host Routines programming models.

Several of the following input functions can read data from a host source. This allows a VersaCode program to receive data from the host computer. For example, when a VersaCode program starts, it may want to receive the latest valid part numbers for an inventory application. (The data sent should use only the printable ASCII character set.) A carriage return is used to mark the end of the host data command packet, and therefore can't be used in the data stream itself.

When host data is sent to the scanner, it is stored to a buffer. If there is already data in the buffer that hasn't yet been read, that data is overwritten with the new data. The maximum amount of data that may be sent at one time is 228 bytes.

Host Input Send Format

The format for a host data command is:

```
<SYN>NS@,HDAT$<CR>
```

Where	<SYN>	is <SYN> is control character (ASCII 22 decimal)
	N	"N" is a non-menu command for ST2070
	S	is always a prefix to scanner ID
	@	is the scanner ID
	$	is the data
	<CR>	is control character (ASCII 13 decimal)

The scanner ID is required on cordless scanners. The comma is required to mark the end of the ID, even when no ID is used. The following example shows a host InputString command represented as a 'C' language string sent to a scanner named "TEST":

```
"\026NSTEST,HDATabcdefg1234567\015"
```

When the following program is executed on the scanner, "abcdefg1234567" is read into string S1:

```
StringS1*300
InputString(S1, Source=Host)
```

See *Built-In Functions* beginning on page 4-1 for details on reading data from the host.

Host Input Status Return Codes

When a host data command is sent by the host to a scanner, the scanner acknowledges receipt of the command. The format for this response is:

```
S@,1,RCL,RC
```

Where

S	is always a prefix to scanner ID
@	is the scanner ID sent with the host data command (if any)
RCL	is the return code length in bytes
RC	is the numeric return code.

The following status codes may be returned:

1	successful receipt
-1	successful receipt, but unread data was overwritten to make room for new data
-2	amount of data is too large
-200	scanner ID is valid, but the scanner did not respond. Possibly due to out of range.
-201	scanner ID is not valid

The following is an example of a response to the host data command example, when there is unread host data on the scanner:

```
STEST,1,2,-1
```

Host Command Mode

In addition to the self-contained VersaCode programs, VersaCode supports host computer-based programs. Host Commands can be used by a host program to control input devices, display output, and communication. Refer to *Built-In Functions* beginning on page 4-1 for Host Commands and syntax.

When a Host Command is sent to the scanner, it is queued until the script asks for it (see Switching to Host Command Mode on page 3-5). Only one Host Command may be queued at a time. When a second Host Command is sent before the first is processed, the second is lost.

Switching to Host Command Mode

A cordless scanner that supports VersaCode is always running a VersaCode program. There is even a default VersaCode program that runs when no other program is selected. A VersaCode program cannot normally be interrupted by a Host Command. In order to allow processing of Host Commands, the VersaCode program running on the cordless scanner must explicitly enable execution of Host Commands using CheckForCommand (see page 4-11), WaitForCommand (see page 4-64), or WaitForMultipleCommands (see page 4-67).

- CheckForCommand executes a single queued Host Command. If no command is queued, the VersaCode program continues.
- WaitForCommand executes a single Host Command. If no command is queued, the cordless scanner waits for one.
- WaitForMultipleCommands waits for and executes Host Commands repeatedly until an END Host Command is received.

Host Command Send Format

The format for a Host Command is:

```
<SYN>NS@,HCMD$<CR>
```

Where	<SYN>	is <SYN> is control character (ASCII 22 decimal)
	N	"N" is a non-menu command for ST2070
	S	is always a prefix to scanner ID
	@	is the scanner ID
	$	is the Host Command
	<CR>	is control character (ASCII 13 decimal)

The scanner ID is required on cordless scanners, but is optional on corded acquisition devices. The comma is required to mark the end of the ID, even when no ID is used. The following example shows a host InputString command represented as a 'C' language string sent to a scanner named "TEST":

```
\026NSTEST,HCMDInputString('Keypad')\015
```

where \026 is the 3-digit octal representation for <SYN> and \015 is the 3-digit octal representation for <CR>

For further information, see the Host Command Compatibility for InputString, page 4-36.

Host Command Receive Format

The scanner will send a response to each Host Command. By default it will put a suffix of a <CR> at the end of each Host Command . To disable the suffix, clear the suffix associated with HHP Code ID 7. For additional information, please refer to the IT3870/75/5770 System Manual.

Good Response

The format for a good Host Command response is:

`S@,N,LA1,A1,LA2,A2,…,LAn,An`

Where
- S is always a prefix to scanner ID
- @ is the scanner ID sent with the Host Command (if any)
- N is the number of return arguments, which varies for each Host Command
- LAn is the length of the return argument n
- An is the contents of return argument n

The following example shows a good response to the InputString Host Command example above, where keypad input “Input” was received:

`STEST,3,1,1,4,k]Z0,5,Input`

Where
- S is always a prefix to scanner ID
- TEST is the scanner ID sent in the InputString example, above
- 3 is the total number of arguments to be returned
- 1 the length of the first argument is 1
- 1 its value is 1, which indicates the operation was a success
- 4 is the length of the second argument
- k]Z0 its value is k]Z0, which indicates the data source as a keypad
- 5 is the length of the final argument
- Input its value is “Input”

For further information, see the Host Command Compatibility for InputString, page 4-36.

Bad Response

The format for a bad Host Command response is:

`S@,EC<CR>`

Where S is always a prefix to scanner ID
- @ is the scanner ID sent with the Host Command (if any)
- EC is the negative numeric error code returned (see Host Command Error Codes, page 3-7)
- <CR> a carriage return terminates the response

The following example shows a response to to an invalid host command sent to scanner "TEST":

STEST,-1<CR>

Where S is always a prefix to scanner ID.

TEST is the scanner ID sent in the InputString example, above.

-1 is the error code for unrecognized Host Command

<CR> a carriage return terminates the response

Host Command Error Codes

The following error codes may be returned as a result of a parsing or execution error on a Host Command:

-1	unrecognized Host Command
-2	extraneous characters found after a valid Host Command
-3	the command is missing a left parenthesis
-4	the command is missing a right parenthesis
-5	the command is missing a string parameter
-6	the command is missing a numeric parameter
-7	the command is missing a label descriptor for a parameter
-8	the command is missing a comma
-9	there is not enough memory available to run the command because the resident script has depleted memory resources
-10	the host routine name is not recognized
-11	the command contains an extraneous comma
-200	scanner ID is valid, but the scanner did not respond. Possibly due to out of range.
-201	scanner ID is not valid

Host Command Examples

In each command, the four-character scanner ID is represented by "SCAN." Control characters are show as a mnemonic in square brackets (e.g., <SYN> represents ASCII decimal 22, and <CR> represents ASCII decimal 13.

EraseDisplay

<SYN>NSSCAN,HCMD**EDSP**<CR>

SSCAN,1,1,1

EraseLine

<SYN>NSSCAN,HCMD**ELN**<CR>

SSCAN,1,1,1

LoadImage

<SYN>NSSCAN,HCMD**LDG(21,26,2)**<CR>

SSCAN,1,1,1

MoveDisplayCursorTo

<SYN>NSSCAN,HCMD**CUR(33,6)**<CR>

SSCAN,1,1,1

OutputString

<SYN>NSSCAN,HCMD**OUTS("Hello",INV)**<CR>

SSCAN,1,1,1

ScrollDisplay

<SYN>NSSCAN,HCMD**SCRL(-8)**<CR>

SSCAN,1,1,1

DisplayMenu

<SYN>NSSCAN,HCMD**DMN(TITL="Is this correct?",MNIT="1.Yes"(HOTK="1"),MNIT="2.No" (HOTK="2"),WRP)**<CR>

SSCAN,1,1,2

InputString

<SYN>NSSCAN,HCMD**INS()**CR]

SSCAN,3,2,16,4,b]A0,16,6625-99-000-0004

InputNumber

<SYN>NSSCAN,HCMD**INN(KPD,AKY)**<CR>

SSCAN,3,1,3,4,k]Z0,3,250

WaitForKey

<SYN>NSSCAN,HCMD**WAITKY**<CR>

SSCAN,1,2,18

End

<SYN>NSSCAN,HCMD**END**<CR>

SSCAN,1,1,1

Joined Scanners

Use the Joined Scanners Host Command to query the base to determine which scanners are attached, where BASE is used as the scanner ID for the base. Although any four characters will suffice, BASE is recommended.

<SYN>NSBASE,HCMDJOINEDSCANNERS<CR>

The response follows the same format as the others, leading with a scanner ID, then the number of scanners that are joined, then a pair of responses for each scanner: number of characters in the scanner ID (always four in the case of the IT3875) and the scanner ID itself. Each data element is separated by a comma. For example, if you have two scanners attached, you would receive:

SBASE,2,4,B515,4,CAD8

If scanners are attached but out of radio contact, they will still be reported. To remove a scanner from the list, you must use the scanner to be removed and scan the "Remove Scanner" bar code below.

Remove Scanner

Host Routines

One of the available Host Commands is Execute, which invokes a routine in the program being executed on the scanner. The routine must be marked as a host routine by including the keyword "HOST" before the routine definition (see example below). A 'Host Routine' can be executed by the host or by the internal script. By default, CheckForCommand (page 4-11), WaitForCommand (page 4-64), and WaitForMultipleCommands (page 4-67) allow any host routine to be executed while in Host Command mode. See the definitions of each of these commands for other means of controlling what is executed. The format of the Host Command Execute is the same as an Execute within a compiled program with two exceptions, taking the following form:

```
Execute(RoutineName [, DataString] [, BeginACK] [,
EndACK])
```

Like the Execute statement (see Execute Statement, page 2-22), the first parameter is the required routine name and the second parameter is an optional string constant that, if present, is loaded into LastDecodedMessage. The two new parameters are BeginACK, which indicates that a boolean ACK (1 for success, 0 for failure) is to be sent to acknowledge command receipt, and EndACK, which indicates that a boolean ACK (1 for success, 0 for failure) is to be sent after successfully running the routine. Other error return codes may be sent back in place of an ending ACK, including a code indicating that the routine name could not be executed (see Host Command Error Codes, page 3-7).

The following is a sample program with a host routine:

```
Host Routine DisplayTest
        EraseDisplay
        OutputString("Display test")
        Wait(1000)
End

Program
        WaitForCommand
End
```

Compile and write this program. As in previous examples, assume the scanner ID is "TEST". The following is an example of a host Execute command represented as a 'C' language string sent to a scanner named "TEST":

```
"\026NSTEST,HCMDExecute(DisplayTest, BeginACK)\015"
```

where \026 is the 3-digit octal representation for <SYN> and \015 is the 3-digit octal representation for <CR>

This Host Command causes the routine DisplayTest to be run. The text "TEST,1,1,1" is returned before actually running DisplayTest, and indicates the command was successfully received (BeginACK).

4 Built-In Functions

Introduction

In order to allow the VersaCode programmer to perform user input / output operations, access system resources such as the timer, and make programming easier, we've included some built-in functions in VersaCode. The following listing provides a detailed description of each routine, accompanied by examples where appropriate.

Conventions

- Required parameters are shown in parentheses ( ).
  Optional parameters are shown in square brackets [ ].
  String and numeric expressions are noted in italics for readability only.
- When entering a function with a series of parameters, we recommend using all the parameters, spelled out as they appear. For example:

```
OutputString(String="Test", Video=Normal,
Font=VeryClearMedium,
SelectForHorizontalScrolling, LeftJustify)
```

  You must follow the correct order of the parameters, as shown in this manual.

  If you choose to leave out some parameters, you must also leave out the commas for those parameters. For example:

```
OutputString("Test",VeryClearMedium)
```

  If you leave in the commas, you must also enter the corresponding parameter or the script will not compile.

- When entering text, hex codes may be used. The format is "\nn" where nn stands for the digits of the hex code. If entering more than one hex code, the format is "\nn\nn". You must enclose hex code entries in quotes. See IT3875 Font Characters Chart, page 8-4, for the hex codes for ASCII characters. When a backslash is encountered in a string constant, it is assumed a hex code will follow. To specify a backslash character, use either the hex code for backslash (\5C), or use 2 backslashes.
- Minimum and maximum values are noted where applicable. However, these are *recommended* values. In some cases, you may enter values outside the minimum and maximum ranges.

BatteryStrength

This function returns the battery strength on a scale of 0 (discharged) to 100 (fully charged).

Syntax

BatteryStrength

Return Value

Returns the battery strength on a scale of 0 (discharged) to 100 (fully charged).

Remote Host Command Compatability

Syntax is same as above.
Host Command: BATLEV

Example

In this example, the battery strength is written to the display.

```
Comment Battery Strength Sample Program
OutputString("Battery strength =
"+string(BatteryStrength))
```

Beep

This function sounds the beeper.

Syntax

Beep(
[Frequency =]*user-defined numeric expression*,
[Duration =]*user-defined numeric expression*,
[Volume =]*user-defined numeric expression*
)

Parameters

Frequency=

This specifies the beep frequency in cycles per second (Hz).
Minimum value: 65
Maximum value: 8000

Duration=

This specifies the beep duration in milliseconds. If the duration exceeds 1 second, the beeper is sounded for 1 second.
Minimum value: 0
Maximum value: 1000

Volume=

Valid volume values are 0 for off, 1 for low, 2 for medium, and 3 for high.

Minimum value: 0

Maximum value: 3

Return Value

None.

Remote Host Command Compatability

Syntax: Beep(frequency, duration, volume), where frequency, duration, and volume are numeric constants. Returns 1 for true if the command is correct, and an error code otherwise.
Host Command: BEP

Example

In this example, the beeper sounds for 100 milliseconds at 800 Hz high volume.

```
Comment Beep Sample Program
Beep(800, 100, 3)
```

CaseInsensitiveCompare

This function compares two strings, ignoring the case.

Syntax

CaseInsensitiveCompare(
user-defined string 1 operator *user-defined string 2*
)

Parameters

user-defined string 1 operator user-defined string 2

Compares user-defined string 1 to user-defined string 2 in a case insensitive manner using an operator (see Logical Expressions on page 2-15).

Return Value

Boolean true if the strings compare according to the operator, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

In this example, the first OutputString call is made.

```
Comment CaseInsensitiveCompare Sample Program
String  S1*32

S1="AbCdEfG"
If(CaseInsensitiveCompare(S1 < "abcdefgh"))
{
OutputString("S1 less than abcdefgh")
}
else
{
OutputString("S1 greater than or equal  to abcdefgh")
}
```

CaseInsensitiveWildcardCompare

This function compares two strings, using wildcards “*”, “#”, and “?” in the first string. The string expressions are compared in a case insensitive manner.

Syntax

CaseInsensitiveWildcardCompare(
user-defined string 1,
user-defined string 2
)

Parameters

user-defined string 1

A string expression that specifies the first string to compare. Wildcard characters “*”, “#”, and “?” may be used in user-defined string 1. A “*” in user-defined string 1 indicates that zero or more characters in the user-defined string 2 can be ignored. A “?” in user-defined string 1 indicates that the next character in user-defined string 2 can be any character. A “#” in user-defined string 1 indicates that the next character in user-defined string 2 must be numeric.

If you are searching for a question mark (?), pound sign (#), or asterisk (*) character in a string, enter the search beginning with a caret (^). For example, to search for "A?B*C," enter the search string "A^?B^*C." If you are searching for a caret mark, use two marks (^^). For example, to search for "A^B," enter the search string "A^^B."

user-defined string 2

The user-defined string to which user-defined string 1 is compared.

Using “abcdefg” as user-defined string 2, the following examples produce the accompanying results:
a user-defined string 1 value of “a*ef*” produces a match
a user-defined string 1 value of “a?cdef?” produces a match
a user-defined string 1 value of “a?bcdefg?” does not produce a match (there must be exactly one character corresponding to the ‘?’)

Return Value

Boolean true if the strings compare, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

In this example, the first and fourth OutputString calls are made.

```
Comment CaseInsensitiveWildcardCompare Sample Program
String  S1*32

S1="AbCdEfG"
If(CaseInsensitiveWildcardCompare("a*ef*",S1))
{
OutputString("S1 equals a*ef*")
}
else
{
OutputString("S1 doesn’t equal  a*ef*")
}
If(CaseInsensitiveWildcardCompare("a*ef??",S1))
{
OutputString("S1 equals a*ef??")
}
else
{
OutputString("S1 doesn’t equal  a*ef??")
}
```

CaseSensitiveCompare

This function compares two strings, in a case-sensitive manner.

Syntax

CaseSensitiveCompare (
user-defined string 1 operator *user-defined string 2*
)

Parameters

user-defined string 1 operator user-defined string 2

Compares user-defined string 1 to user-defined string 2 in a case sensitive manner using an operator (see Logical Expressions on page 2-15).

Return Value

Boolean true if the strings compare according to the operator, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

In this example, the second OutputString call is made.

```
Comment CaseSensitiveCompare Sample Program
String
 S1*32

S1="AbCdEfG"
If(CaseSensitiveCompare (S1 < "ABCDEFGH"))
{
OutputString("S1 less than ABCDEFGH")
}
else
{
OutputString("S1 greater than or equal  to ABCDEFGH")
}
```

CaseSensitiveWildcardCompare

This function compares two strings, using wildcards “*”, “#”, and “?” in the first string. The string expressions are compared in a case sensitive manner.

Syntax

CaseSensitiveWildcardCompare(
user-defined string 1,
user-defined string 2
)

Parameters

user-defined string 1

A string expression that specifies the first string to compare. Wildcard characters “*”, “#”, and “?” may be used in user-defined string 1. A “*” in user-defined string 1 indicates that zero or more characters in user-defined string 2 can be ignored. A “?” in user-defined string 1 indicates that the next character in user-defined string 2 can be any character. A “#” in user-defined string 1 indicates that the next character in user-defined string 2 must be numeric.

If you are searching for a question mark (?), pound sign (#), or asterisk (*) character in a string, enter the search beginning with a caret (^). For example, to search for "A?B*C," enter the search string "A^?B^*C." If you are searching for a caret mark, use two marks (^^). For example, to search for "A^B," enter the search string "A^^B."

user-defined string 2

The user-defined string to which user-defined string 1 is compared.

Using “abcdefg” as user-defined string 2, the following examples produce the accompanying results:
a user-defined string 1 value of “a*ef*” produces a match
a user-defined string 1 value of “a?cdef?” produces a match
a user-defined string 1 value of “a?bcdefg?” does not produce a match (there must be exactly one character corresponding to the ‘?’)

Return Value

Boolean true if the strings compare, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

In this example, the second and fourth OutputString calls are made.

```
Comment CaseSensitiveWildcardCompare Sample Program
String  S1*32

S1="AbCdEfG"
If(CaseSensitiveWildcardCompare("a*ef*",S1))
{
OutputString("S1 equals a*ef*")
}
else
{
OutputString("S1 doesn't equal  a*ef*")
}
If(CaseSensitiveWildcardCompare("a*ef??",S1))
{
OutputString("S1 equals a*ef??")
}
else
{
OutputString("S1 doesn't equal  a*ef??")
}
```

CharacterHeight

This function gets the height of the characters, in pixels.

Control characters (00-1F Hex) take up two times the character width. All other characters take one character width. Refer to the table below for the font size in pixels.

Font	*Size in Pixels*
VeryClearSmall	6w x 8h
VeryClearMedium	8w x 11h
VeryClearLarge	12w x 16h

Syntax

CharacterHeight(
[VeryClearSmall | VeryClearMedium | VeryClearLarge]
)

Parameters

VeryClearSmall | VeryClearMedium | VeryClearLarge

An integer or a font name may be specified.
0 or VeryClearSmall
1 or VeryClearMedium
2 or VeryClearLarge

Minimum value: 0
Maximum value: 2
Default: The active font

Return Value

Returns the integer height of a character in pixels.

Remote Host Command Compatability

Syntax is same as above. Returns the character height. When an integer is used instead of a font name, the number must be a constant.
Host Command: CHH

Host Parameters:		
	VeryClearSmall	VCS or 0
	VeryClearMedium	VCM or 1
	VeryClearLarge	VCL or 2

Example

In this example, the character height of the current active font is written to the display, followed by the character height of VeryClearSmall.

```
Comment CharacterHeight Sample Program
OutputString("Character height =
"+string(CharacterHeight))
OutputString("Character height =
"+string(CharacterHeight(VeryClearSmall)))
```

CharacterWidth

This function gets the width of the characters, in pixels.

Control characters (00-1F Hex) take up two times the character width. All other characters take one character width. Refer to the table below for the font size in pixels.

Font	*Size in Pixels*
VeryClearSmall	6w x 8h
VeryClearMedium	8w x 11h
VeryClearLarge	12w x 16h

Syntax

CharacterWidth(
[VeryClearSmall | VeryClearMedium | VeryClearLarge],
user-defined string expression
)

Parameters

VeryClearSmall | VeryClearMedium | VeryClearLarge

An integer or a font name may be specified.
0 or VeryClearSmall
1 or VeryClearMedium
2 or VeryClearLarge

Minimum value: 0
Maximum value: 2
Default: The active font

user-defined string expression

A string expression that specifies the character whose width is to be determined. Note that the string expression must be exactly one character.

Return Value

Returns the integer width in pixels of the character in the string expression. When an integer is used instead of a font name, the number must be a constant.

Remote Host Command Compatability

Syntax: same as above, except that the character is enclosed in single quotes. Returns the character width.
Host Command: CHW

Host Parameters:		
	VeryClearSmall	VCS or 0
	VeryClearMedium	VCM or 1
	VeryClearLarge	VCL or 2

Example

In this example, the character width of the letter ‘a’ for both the active font and for VeryClearLarge is written to the display.

```
Comment CharacterWidth Sample Program
OutputString("Character width of ‘a’ =
"+string(CharacterWidth(VeryClearLarge, "a")))
```

CheckForCommand

This function checks to see if a host VersaCode command has been received. If it has, it is processed and control is returned to the VersaCode program. Only one command can be processed.

Syntax

CheckForCommand(
[[HostRoutines=]NONE | HOST | *(list)*]
)

Parameters

NONE | HOST | string expressions

NONE	No routines can be executed by the host while in host command mode.
HOST	Only host routines can be executed by the host.
list	String expressions specifying the names of routines that can be executed by the host. For example, CheckForCommand(HostRoutines=("AB","CD")) indicates that the host can execute host routines AB and CD.

Default: HOST

Return Value

If a host routine is executed via the host command Execute, the routine name is returned as a string. Otherwise, a null string is returned.

Remote Host Command Compatability

Not compatible.

Example

In this example, the first OutputString is executed. The VersaCode processor checks to see if a host VersaCode command has been received. If so, the host VersaCode command is processed, then the second OutputString is executed.

```
Comment CheckForCommand Sample Program
OutputString("Check for a VersaCode command")
CheckForCommand
OutputString("A VersaCode command may have been
executed")
```

DefaultPrefix

This function sets a default prefix that can be used by the Send function.

Syntax

DefaultPrefix (
user-defined string expression,
[Always | IfData | IfNoData]
)

Parameters

user-defined string expression

A string expression that specifies the default prefix.

Always | IfData | IfNo Data

Sets a default prefix that can be used by the Send function depending on whether or not the string in the Send statement is empty (no data). This statement is not used as a prefix command within the Send statement, but defines the default prefix which can be used by the Send statement. Default: Always.

Return Value

None.

Remote Host Command Compatability

Not compatible.

Example

This code defines a default prefix, whose value is “Prefix”, that should be applied when there is no data in the Send statement string expression. Since the Send statement is trying to write null data to the host, the default prefix is applied.

```
Comment DefaultPrefix Sample Program
DefaultPrefix("Prefix", IfNoData)
Send("", DefaultPrefix)
```

DefaultSuffix

This function sets a default suffix that can be used by the Send function.

Syntax

DefaultSuffix (
user-defined string expression,
[Always | IfData | IfNoData]
)

Parameters

user-defined string expression

A string expression that specifies the default suffix.

Always | IfData | IfNo Data

Sets a default suffix that can be used by the Send function depending on whether or not the string in the Send statement is empty (no data). This statement is not used as a suffix command within the Send statement, but defines the default suffix which can be used by the Send statement. Default: Always.

Return Value

None.

Remote Host Command Compatability

Not compatible.

Example

This code defines a default suffix, whose value is “Suffix”, that should be applied when there is no data in the Send statement string expression. Since the Send statement is trying to write null data to the host, the default suffix is applied.

```
Comment DefaultSuffix Sample Program
DefaultSuffix("Suffix", IfNoData)
Send("", ,DefaultSuffix)
```

DisplayBacklightOff

This function turns off the backlight for the display.

Note: This function can be overridden if a user selects Backlight Control - On using the scanner keypad. This function is enabled when a user selects Backlight Control - Script using the scanner keypad.

Syntax

DisplayBacklightOff

Return Value

None.

Remote Host Command Compatability

Syntax: DisplayBacklightOff. Returns 1 for true if the command was correct, and an error code otherwise.
Host Command: BKD

Example

In this example, the display backlight is turned off between the first and second OutputString.

```
Comment DisplayBacklightOff Sample Program
DisplayBacklightOn
OutputString("Backlight is on")
Wait(1000)
DisplayBacklightOff
OutputString("Backlight is off")
```

DisplayBacklightOn

This function turns on the backlight for the display.

Syntax

DisplayBacklightOn

Return Value

None.

Remote Host Command Compatability

Syntax: DisplayBacklightOn. Returns 1 for true if the command was correct, and an error code otherwise.
Host Command: BKE

Example

In this example, the display backlight is turned on between the first and second OutputString.

```
Comment DisplayBacklightOn Sample Program
DisplayBacklightOff
OutputString("Backlight is off")
Wait(1000)
DisplayBacklightOn
OutputString("Backlight is on")
```

DisplayCursorOff

This turns the display cursor off. This function does not affect InputString (see page 4-36) and InputNumber (see page 4-31) operations.

Syntax

DisplayCursorOff

Return Value

None.

Remote Host Command Compatability

Returns 1 for true if the command was correct, 0 otherwise.
Host Command: CURD

Example

In this example, the cursor is turned off. No cursor appears after the word "Testing" is displayed.

```
Comment DisplayCursorOff Sample Program
DisplayCursorOff
OutputString("Testing")
```

DisplayCursorOnBlink

This function turns the display cursor on to a blinking state. This does not affect InputString (see page 4-36) and InputNumber (see page 4-31) operations.

Syntax

DisplayCursorOnBlink

Return Value

None.

Remote Host Command Compatability

Returns 1 for true if the command was correct, 0 otherwise.
Host Command: CURB

Example

In this example, the cursor is turned on to a blinking state. A blinking cursor appears after the word “Testing” is displayed.

```
Comment DisplayCursorOnBlink Sample Program
DisplayCursorOnBlink
OutputString("Testing")
```

DisplayCursorOnSteady

This function turns the display cursor on to a steady state. This does not affect InputString (see page 4-36) and InputNumber (see page 4-31) operations.

Syntax

DisplayCursorOnSteady

Return Value

None.

Remote Host Command Compatability

Returns 1 for true if the command was correct, 0 otherwise.
Host Command: CURS

Example

In this example, the cursor is turned on to a steady state. A steady cursor appears after the word “Testing” is displayed.

```
Comment DisplayCursorOnSteady Sample Program
DisplayCursorOnSteady
OutputString("Testing")
```

DisplayHeight

This function returns the height of the output display, in pixels.

Syntax

DisplayHeight

Return Value

Returns the integer height of the output display in pixels.

Remote Host Command Compatability

Syntax: DisplayHeight. Returns the height of the display.
Host Command: DSPH

Example

In this example, the display height is written to the display.

```
Comment DisplayHeight Sample Program
OutputString("Display height =
"+string(DisplayHeight))
```

DisplayLanguage

DisplayLanguage returns the current selected display language used.

Syntax

DisplayLanguage

Return Value

Returns one of the following strings: “English,” “Spanish,” “Dutch,” “French,” “German,” “Italian,” or “Unknown.”

Remote Host Command Compatability

Syntax is the same as above. The returned parameter is a string whose possible values are defined in Return Value.
Host Command: DSPLNG

Example

In this example, the currently selected language is written to the display.

```
Comment DisplayLanguage Sample Program
OutputString(DisplayLanguage)
```

DisplayMenu

DisplayMenu constructs a text menu on the display and waits for a selection.

Syntax

DisplayMenu (
[MenuStart=*(user-defined numeric expression x*, *user-defined numeric expression y)*,]
[Title=*user-defined string expression 1*[([No]Scroll)],]
[MenuItem=*string expression 2* [([Hotkey=*user-defined string expression 3*,] [DefaultSelection])],]
[WrapOnSelectionScrolling,]
[MaximumDisplayLinesToUse=*user-defined numeric expression 3*,]
[TimeoutOnNoSelection=*user-defined numeric expression 4*,]
[QuitOnBackspace]
)

Parameters

MenuStart=

The upper left corner of the menu is positioned at the menu start coordinates specified by user-defined numeric expressions x and y.
Minimum value: 0, 0

Maximum value: display width, display height

Default: 0,0

Title=

An optional title (user-defined string expression 1) may be provided. If present, it may be configured to scroll with the rest of the menu on the display (Scroll), or to stay in the same location during scrolling (NoScroll).

Default: No title (NoScroll will be the default if a title is used, but no scroll setting is specified.)

MenuItem=

The menu may consist of 0 to 15 menu item specifiers, where each one is a menu selection line in ascending order down the display. Each menu item must have a user-defined string expression to identify it.

Hotkey	An optional hot key may be specified for the menu item. Default: No Hotkey
DefaultSelection	This indicates that this menu item is the highlighted default selection. If more than one menu item is marked as the default, the last menu item so marked is the default selection. Default: The first menu item is the default selection if no item is specified as the default selection.

WrapOnSelectionScrolling

When scrolling with the up or down arrows, scrolling may be configured to wrap from the last menu item back to the first and vice versa. Default: No WrapOnSelectionScrolling

MaximumDisplayLinesToUse

The maximum number of display lines to use may be specified, if you wish to use only part of the screen to display the menu. The menu would then have to be scrolled. Default: The entire display

TimeoutOnNoSelection

A time out (in milliseconds) may be specified to indicate how long to wait without user input before terminating the display menu operation.

Minimum value: 0

Maximum value: 64000 (64 seconds)

Default: No time out

QuitOnBackspace

Backspace may be used as an escape character to exit the menu.
Default: No QuitOnBackspace

Return Value

If DisplayMenu is part of a user-defined numeric expression, then it returns either the item number selected in the range of 1 to the number of items, 0 if backspace was pressed, and –1 if the operation timed out.

If DisplayMenu is part of a user-defined string expression, it returns either the menu item string that was selected, the backspace character, if backspace was pressed, or a null string if the operation timed out.

Remote Host Command Compatability

Syntax is same as above, except that string expressions must be string constants, numeric expressions must be numeric constants, and the return code is always numeric.
Host Command: DMN

Host Parameters:	MenuStart	MNST
	Title	TITL
	NoScroll	SCLN
	Scroll	SCLY
	MenuItem	MNIT
	HotKey	HOTK
	DefaultSelection	DEF
	WrapOnSelectionScrolling	WRP
	MaximumDisplayLinesToUse	MXLN
	TimeoutOnNoSelection	TMOUT
	QuitOnBackspace	BSQ

Example

In the following example, the first DisplayMenu returns the string selected. It times out after 5 seconds if no input is received. It requires vertical scrolling because the menu requires 4 lines, but only 3 display lines are allocated. It starts at position (10,10) on the display. The second DisplayMenu returns the item number selected.

```
Comment DisplayMenu Sample Program
Int     I1
String  S1*32
```

```
S1 = DisplayMenu (MenuStart=(10,10), Title="Test
Menu"(NoScroll),
MenuItem="Item 1"(Hotkey="a", DefaultSelection),
MenuItem="Item 2"(Hotkey="b"),
MenuItem="Item 3"(Hotkey="b"),
WrapOnSelectionScrolling,
MaximumDisplayLinesToUse= 3,
TimeoutOnNoSelection=5000,
QuitOnBackspace)

I1 = DisplayMenu (MenuItem="Item 1"(Hotkey="a",
DefaultSelection),
MenuItem="Item 2"(Hotkey="b"))
```

DisplayWidth

Gets the width of the output display in pixels.

Syntax

DisplayWidth

Return Value

Returns the integer width of the output display in pixels.

Remote Host Command Compatability

Syntax: DisplayWidth. Returns the width of the display.
Host Command: DSPW

Example

In this example, the display width is written to the display.

```
Comment DisplayWidth Sample Program
OutputString("Display width = "+string(DisplayWidth))
```

EraseDisplay

Erases the entire display, leaving the cursor in the upper left corner.

Syntax

EraseDisplay

Return Value

None.

Remote Host Command Compatability

Syntax: EraseDisplay. Returns 1 for true if the command was correct, and an error code otherwise.
Host Command: EDSP

Example

In this example, “Before erasing” is written to the display, the display is erased, then “After erasing” is written to the display.

```
Comment EraseDisplay Sample Program
OutputString("Before erasing")
EraseDisplay
OutputString("After erasing")
```

EraseLine

Erases a line in the display, starting at the current cursor position.

Syntax

EraseLine(
[[x=]*user-defined numeric expression 1*,
[y=]*user-defined numeric expression 2*]
)

Parameters

x=

The number of columns to erase from the start point is specified by this numeric expression. If the column and row counts are not present, then starting from the current cursor position, the remainder of the row of one character height (according to the active font) is erased. The cursor remains at the upper left corner of the erased line.
Minimum value: 1
Maximum value: Number of display columns from the current cursor position
Default: Number of display columns from the current cursor position

y=

The number of rows to erase from the start point is specified by this numeric expression. If the column and row counts are not present, then starting from the current cursor position, the remainder of the row of one character height (according to the active font) is erased. The cursor remains at the upper left corner of the erased line.
Minimum value: 1
Maximum value: One character height according to the active font
Default: One character height according to the active font

Return Value

None.

Remote Host Command Compatability

Syntax: EraseLine(x,y), where x and y are numeric constants specifying the width and length as defined in the description above. Returns 1 for true if the command was correct, and an error code otherwise.
Host Command: ELN

Example 1

In the following example, assuming the current cursor location is at (1,1), the area erased is a 10 by 10 pixel area from (1,1) to (10,10).

```
Comment EraseLine Sample Program
EraseLine(10, 10)
```

Example 2

```
Comment shorthand to erase from cursor to end using
character height.
EraseLine()
```

Find

Finds string 1 in string 2.

Syntax

[Wildcard]Find(
[SearchString=]*user-defined string expression 1*,
[StringToSearch=]*user-defined string expression 2*,
[[SearchStartPosition=]*user-defined numeric expression 1*],
[CaseSensitive | CaseInsensitive],
[Forward | Backward],
[[StringPosition=]*user-defined numeric variable*]
)

Parameters

SearchString=

The string expression to search for. If the keyword *Wildcard* appears at the front of the word *Find*, then wildcard search characters "*", "#", and "?" are used in string expression 1 to search for an occurrence of string expression 1 in string expression 2. A '?' in user-defined string expression 1 indicates that the next character in user-defined string expression 2 can be any character. A '*' in user-defined string expression 1 indicates that zero or more characters in user-defined string expression 2 can be ignored. A "#" in user-defined string 1 indicates that the next character in user-defined string 2 must be numeric.

If you are searching for a question mark (?), pound sign (#), or asterisk (*) character in a string, enter the search beginning with a caret (^). For example, to search for "A?B*C," enter the search string "A^?B^*C." If you are searching for a caret mark, use two marks (^^). For example, to search for "A^B," enter the search string "A^^B."

StringToSearch=

The user-defined string to which search string 1 is compared.

SearchStartPosition=

The starting point may be set by this numeric expression. The default starting point of the search is at the beginning or end, depending on whether the search is forward or backward.

Minimum value: 0

Maximum value: Number of characters in the string to search, minus 1

Default:0 for forward search
Number of characters in the string to search, minus 1, for backward search

CaseSensitive | CaseInsensitive

Alphabetic character comparisons are according to the default case sensitivity unless otherwise explicitly specified.

Default: Default case sensitivity

Forward | Backward

The search proceeds forward by default, unless otherwise specified.

Default: Forward

StringPosition=

A user-defined numeric variable may be loaded with the position where the string was actually found.

Using “abcdefg” as the string to be compared, the following examples produce the accompanying results:
a search string of “a*ef*” produces a match
a search of “a?bcdef?” produces a match
a search string of “a?bcdefg?” does not produce a match (there must be exactly one character corresponding to the ‘?’)

Return Value

Returns the integer position where string 1 was found in string 2; a –1 is returned if the string is not found.

Remote Host Command Compatability

Not compatible.

Example

In this example, the “Find position is 2” is written to the display.

```
Comment Find Sample Program
String  S1*32
Int     I1

S1="AbCdEfG"
I1 = Find("cde", S1, CaseInsensitive)
```

GetFont

Gets the index value of the active font.

Control characters (00-1F Hex) take up two times the character pixel width. All other characters take one character pixel width. Refer to the table below for the font size in pixels.

Font	***Size in Pixels***
VeryClearSmall	6w x 8h
VeryClearMedium	8w x 11h
VeryClearLarge	12w x 16h

Note: The return value must be assigned to a variable or used in an expression.

Syntax

GetFont

Return Value

Returns an integer index value of:

0 for VeryClearSmall
1 for VeryClearMedium
2 for VeryClearLarge

Remote Host Command Compatability

Syntax: GetFont. Returns the index value of the active font.
Host Command: GFNT

Example

In this example, assuming that the current active font is VeryClearLarge, FontIndex is set to 2.

```
Comment GetFont Sample Program
Int     FontIndex

FontIndex = GetFont
```

GetKey

GetKey gets the scan code of the oldest keypress in the keypad buffer and removes it from the buffer. Refer to Numeric Key Codes, page 8-6 for an illustration of the keypad and the number key codes assigned to each key and each FN-key sequence.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

GetKey

Return Value

Returns the integer scan code of the key code read from the keypad buffer, or the key being pressed. If no key was read, 0 is returned. The following keys elicit the following returns:

Key	Returns	Function Key	Returns
1	1	1	65
2	2	2	66
3	3	3	67
4	4	4	68
5	5	5	69
6	6	6	70
7	7	7	71
8	8	8	72
9	9	9	73
0	10	0	74
*	11	down arrow	81
Enter	16	up arrow	82
down arrow	17		
up arrow	18		
backspace	19		

Note: In the case of a FN-key sequence, the FN key is pressed and released and then another key is pressed.

Remote Host Command Compatability

Syntax is same as above. Returns the scan code of the key that was read.

Example 1

In this example, the scan code of the next key in the keypad buffer is written to the display.

```
Comment GetKey Sample Program
OutputString("Key read = "+string(GetKey))
```

Example 2

In this example, we'll look for the "1" key to be pressed. As soon as we see it, we'll go get a string from the keypad, which will include the "1" key as its first character.

```
Comment GetKey Sample Program
Int     I1
String  S1*32

Do  {
        I1 = KeyLook
        If(I1<>1)
        {
        I1 = GetKey
        }
}  While (I1 <> 1)
InputString(S1, Source=Keypad, PreserveBuffer)
```

GetTimer0

Returns the value of a system timer in milliseconds. The timer value reflects the number of elapsed milliseconds since the timer was reset (see ResetTimer0 on page 4-51). After Timer0 counts up to 65,535, it starts again from zero.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

GetTimer0

Return Value

The integer value of Timer0.

Not compatible.

Example

In this example, the trigger state is repeatedly tested for up to 30 seconds. Note that Timer0 counts up to 65,535, "rolls over" to zero, and continues counting. Since the "& 65535" has the effect of clearing the upper bits, it is okay if Timer0 rolls over during the 30 second count.

```
Comment GetTimer0 Sample Program
int iStartTime

EraseDisplay
iStartTime = GetTimer0
while ( ((GetTimer0 - iStartTime) & 65535) < 30000 )
{
 MoveDisplayCursorTo( 0, 0 )
 OutputString( "Trigger=" + string(TriggerHit) )
}
MoveDisplayCursorTo( 0, CharacterHeight )
OutputString( "Done!" )
```

IndicatorBad

This function initiates a bad read beeping and LED flashing sequence (3 beeps and the LED blinks red).

Syntax

IndicatorBad

Return Value

None

Remote Host Command Compatability

Syntax is the same as above. Returns 1 if the command was correct, and an error code otherwise (see Host Routines on page 3-9).
Host Command: IBS

Example

In the following example, the bad LED/beeper sequence is initiated.

```
Comment IndicatorBad Sample Program
IndicatorBad
```

IndicatorGood

This function initiates the good read beeping and LED flashing sequence (1 beep and the LED is steady green).

Syntax

IndicatorGood

Return Value

None

Remote Host Command Compatability

Syntax is the same as above. Returns 1 if the command was correct, and an error code otherwise (see Bad Response on page 3-6).
Host Command: IGS

Example

In the following example, the good LED/beeper sequence is initiated.

```
CommentIndicatorGood Sample Program
IndicatorGood
```

IndicatorLightOff

This function turns off all indicator lights.

Syntax

IndicatorLightOff

Return Value

None.

Remote Host Command Compatability

Syntax is the same as above. Returns 1 if the command was correct, and an error code otherwise (see Bad Response on page 3-6).
Host Command: ILD

Example

In the following example, the green indicator light is turned on and starts the timer for 3 seconds, the unit waits for 200 milliseconds, then the light is forced off.

```
Comment IndicatorLightOff Sample Program
IndicatorLightOn(Green,3000,1)
Wait(200)
IndicatorLightOff
```

IndicatorLightOn

This function turns on the specified indicator light. The light is cycled on and off for a specified time, and for a specified number of cycles.

Syntax

IndicatorLightOn(
[Red | Green | Yellow],
[[OnCycleTime=]*user-defined numeric expression 1*],
[[NumberOfCycles=]*user-defined numeric expression 2*],
)

Parameters

Red | Green | Yellow

The color of the indicator light that will be used. A numeric expression may be used: 1 for Red, 2 for Green, or 3 for Yellow.
Minimum Value: 1
Maximum Value: 3

OnCycleTime=

A user-defined numeric expression specifies the number of milliseconds for the on cycle. The off cycle will use the same time as the on cycle.
Minimum Value: 0
Maximum Value: 30,000

NumberOfCycles=

A user-defined numeric expression specifies the number of cycles.
Minimum Value: 0
Maximum Value: 200

Return Value

None.

Remote Host Command Compatability

Syntax is the same as above, but if a number is used instead of the indicator light name, the number must be a constant. Returns 1 if the command was correct, and an error code otherwise (see Bad Response on page 3-6).

Host Command: ILE

Host Parameters:		
	Red	RED
	Green	GRN
	Yellow	YEL

Example

In the following example, the red indicator light is turned on for 3 seconds, off for 3 seconds, and repeats one more cycle.

```
Comment IndicatorLightOn Sample Program

IndicatorLightOn(Red,3000,2)
```

InputNumber

This command waits for a number from the device.

Note: While the scanner is waiting for input, it is in a lower power consuming mode, as long as the optional timeout parameter is not used.

Note: The InputNumber command clears the LCD display line from the cursor position to the end of the display possibly causing undesired results. For example, if you have a border graphic displayed, a portion of the right side of the border will be erased.

Syntax

InputNumber(
[Number=]*user-defined numeric variable*,
[[Source=] Keypad | Scanner | Host | All],
[[DefaultValue=]*user-defined numeric expression 1*],
[UseArrowKeys | IgnoreArrowKeys],
[QuitOnKey=*user-defined numeric expression 2*],
[ContinueOnNoDecode | QuitOnNoDecode],
[CursorOn | CursorOff],
[SilentInput],
[RefreshDisplayAfterInput],
[FlushBuffer | PreserveBuffer],
[[DataSource=]*user-defined string variable*],
[[Minimum=]*user-defined numeric expression 3*],
[[Maximum=]*user-defined numeric expression 4*],
[[Timeout=]*user-defined numeric expression 5*]
)

Number=

A numeric variable where the number input by the user is to be stored.

Source=

Keypad	Returns a number when the Enter key is hit.
Scanner	Returns a number after a valid read.
Host	A number is accepted from the host (228 max. bytes).
All	A number is accepted from a keypad or scanner or host.

Any combination of source types can be used, separated by '+.' Input Number does not allow non-numeric keys to be entered. Non-numeric characters entered via scanning causes an error beep sequence and are discarded. Default: Source=Keypad+Scanner

DefaultValue=

An optional default number is displayed at the cursor. If Enter is hit prior to any keypad or scanner changes, the default is accepted and returned.

Default: No DefaultValue

UseArrowKeys | IgnoreArrowKeys

UseArrowKeys increments/decrements the number when the up or down arrow is hit.
IgnoreArrowKeys ignores any arrow key presses.
Default: UseArrowKeys

QuitOnKey

Quit keys (one for each possible scan code value) may be defined using QuitOnKey. When a quit key is hit, the InputNumber operation is terminated and the scan code of the key is negated and returned as the return value. Backspace and enter may not be defined as quit keys. If arrow keys are configured for scrolling, they may not be defined as quit keys.
Default: No QuitOnKey

ContinueOnNoDecode | QuitOnNoDecode

ContinueOnNoDecode continues this command when no decode occurs after a trigger pull. QuitOnNoDecode quits and goes to the next command when no decode occurs after a trigger pull.

Default: ContinueOnNoDecode

CursorOn | CursorOff

CursorOn keeps the input cursor on while waiting for input.
CursorOff turns off the input cursor while waiting for input.

Default: CursorOn

SilentInput

This specifies no key click during the InputNumber operation.
Default: No SilentInput

RefreshDisplayAfterInput

This specifies that the display area used during InputNumber is restored to its original content at the end of the operation.
Default: No RefreshDisplayAfterInput

FlushBuffer / PreserveBuffer

PreserveBuffer includes the input stream of any characters in the keypad buffer at the start of the InputNumber operation. FlushBuffer does not.
Default: FlushBuffer

DataSource=

The data source information may optionally be stored to this number variable with the HHP code ID in the first byte, followed by the 3 letter AIM Symbology ID. For additional information, please refer to the IT3870/75/5770 System Manual. Default: No DataSource

Minimum=

A minimum range may be specified for acceptable variable limit to complete command.
Minimum value: 0
Maximum value: specified maximum value, or 999999999, whichever is less
Default: 0

Maximum=

A maximum range may be specified for acceptable variable limit to complete command.
Minimum value: specified minimum value, or 0, whichever is more
Maximum value: 999999999
Default: 999999999

Timeout=

An optional time out may be specified in milliseconds.
Minimum value: 0
Maximum value: 64000 (64 seconds)
Default: No Timeout

Return Value

When InputNumber is part of a Boolean expression, it returns true if a number is found, false otherwise. When InputNumber is part of a numeric expression, it returns the length of the data read, or the negation of a quit key scan code if a quit key was pressed, or:

-10000	quit on no decode
-10001	timeout
-10002	corrupt number entry
-10003	keypad buffer overflow

Key	Returns	Function Key	Returns
1	1	1	65
2	2	2	66
3	3	3	67
4	4	4	68
5	5	5	69
6	6	6	70
7	7	7	71
8	8	8	72
9	9	9	73
0	10	0	74
*	11	down arrow	81
Enter	16	up arrow	82
down arrow	17		
up arrow	18		
backspace	19		

Note: In the case of a FN-key sequence, the FN key is pressed and released and then another key is pressed.

Remote Host Command Compatability

Syntax is the same as above with the following exceptions:

- No integer variable is specified to hold the result, since the result is reported as a parameter to the host.
- No data source string variable is specified, since the data source is always reported to the host
- All numeric expressions must be numeric constants (default value, min value, time out, and max value).
- If the default value and time out parameters are used, the DefaultValue or Timeout label must accompany them, but no other parameter labeling is allowed.
- The returned parameters are a length/status code, the 4 byte data source, and the number that was input.
- HOST is not a valid input source.

The returned parameters are a length/status code (similar to the numeric return value described above), the 4 byte data source, and the actual number that was input.

Host Command: INN

Host Parameters:		
	Keypad	KPD
	Scanner	SCAN
	All	AL
	DefaultValue	DEF
	UseArrowKeys	AKY
	IgnoreArrowKeys	AKN
	QuitOnKey	QKY
	ContinueOnNoDecode	NDC
	QuitOnNoDecode	NDQ
	CursorOn	CURE
	CursorOff	CURD
	SilentInput	INSL
	RefreshDisplayAfterInput	REFR
	FlushBuffer	DMP
	PreserveBuffer	SAV
	Minimum	MIN
	Maximum	MAX
	Timeout	TMOUT

Example

In this example, a number is input from the keypad (the default is 5) and must be in the range of 2 to 10. If successful, the number is displayed.

```
Comment InputNumber Sample Program
Int     I1


I1=5
If(InputNumber(I1, Keypad, DefaultValue=I1,
UseArrowKeys, Minimum=2, Maximum=10))
{
OutputString("Number ="+string(I1))
}
```

InputString

This command waits for a string from the device.

Note: While the scanner is waiting for input, it is in a lower power consuming mode, as long as the optional timeout parameter is not used.

Note: The InputString command clears the LCD display line from the cursor position to the end of the display possibly causing undesired results. For example, if you have a border graphic displayed, a portion of the right side of the border will be erased.

Syntax

InputString(
[String=]*user-defined string pointer,*
[[Source=] Keypad | Scanner | Host | All],
[[DefaultValue=]*user-defined string expression*],
[QuitOnKey=*user-defined numeric expression 2*],
[ContinueOnNoDecode | QuitOnNoDecode],
[CursorOn | CursorOff],
[SilentInput],
[RefreshDisplayAfterInput],
[FlushBuffer | PreserveBuffer],
[[DataSource=]*user-defined string variable*],
[[Minimum=]*user-defined numeric expression 3*],
[[Maximum=]*user-defined numeric expression 4*],
[[Timeout=]*user-defined numeric expression*]
)

Parameters

String=

The InputString function places the returned data in the memory pointed to by this string pointer. Note: It is up to the user to allocate adequate memory for the returned data.

Source=

Keypad	Returns a string when the Enter key is hit.
Scanner	Returns a string after a valid read.
Host	A string is accepted from the host (228 max bytes).
All	A string is accepted from a keypad or scanner or host.

Any combination of source types can be used, separated by '+.' Input Number will error beep if the keypad is enabled and Enter is pressed with no input, or if input does not meet minimum or maximum limits. Default: Source=Keypad+Scanner

DefaultValue=

An optional default string is displayed at the cursor. If Enter is hit prior to any keypad or scanner changes, the default is accepted and returned. Default: No DefaultValue

QuitOnKey

Quit keys (one for each possible scan code value) may be defined using QuitOnKey. When a quit key is hit, the InputString operation is terminated and the scan code of the key is negated and returned as the return value. Backspace and enter may not be defined as quit keys. If arrow keys are configured for scrolling, they may not be defined as quit keys.
Default: No QuitOnKey

ContinueOnNoDecode | QuitOnNoDecode

ContinueOnNoDecode continues this command when no decode occurs after a trigger pull. QuitOnNoDecode quits and goes to the next command when no decode occurs after a trigger pull. Default: ContinueOnNoDecode

CursorOn | CursorOff

CursorOn keeps the input cursor on while waiting for input. CursorOff turns off the input cursor while waiting for input.
Default: CursorOn

SilentInput

This prevents input data from appearing on the display as it's entered. Default: No SilentInput

RefreshDisplayAfterInput

The display area that showed data as it was being entered is restored to its state prior to the input operation.
Default: No RefreshDisplayAfterInput

FlushBuffer / PreserveBuffer

PreserveBuffer includes the input stream of any characters in the keypad buffer at the start of the InputString operation. FlushBuffer does not.
Default: FlushBuffer

DataSource=

The data source information may optionally be stored to this string variable with the HHP code ID in the first byte, followed by the 3 letter AIM Symbology ID. For additional information, please refer to the IT3870/75/5770 System Manual. Default: No DataSource

Minimum=

A minimum range may be specified for the length of the string limit to complete command.
Minimum value: 0
Maximum value: specified maximum value, or 3000 bytes, whichever is less
Default: 1

Maximum=

A maximum range may be specified for the length of the string limit to complete command.
Minimum value: specified maximum value, or 0, whichever is greater
Maximum value: 3000 bytes
Default: 3000 bytes

Timeout=

An optional time out is specified in milliseconds.
Minimum value: 0
Maximum value: 64000 (64 seconds)
Default: No Timeout

Return Value

Returns the length of the data read, or the negation of a quit key scan code if a quit key was pressed, or:

-10000	quit on no decode
-10001	timeout
-10003	keypad buffer overflow

Key	Returns	Function Key	Returns
1	1	1	65
2	2	2	66
3	3	3	67
4	4	4	68
5	5	5	69
6	6	6	70
7	7	7	71
8	8	8	72
9	9	9	73
0	10	0	74
*	11	down arrow	81
Enter	16	up arrow	82
down arrow	17		
up arrow	18		
backspace	19		

Note: In the case of a FN-key sequence, the FN key is pressed and released and then another key is pressed.

Remote Host Command Compatability

Syntax is the same as above with the following exceptions:

- No string variable is specified to hold the result, since the result is reported as a parameter to the host.
- No data source string variable is specified, since the data source is always reported to the host

- All numeric expressions (time out) must be numeric constants.
- If the default value and time out parameters are used, the DefaultValue or Timeout label must accompany them, but no other parameter labeling is allowed.
- The returned parameters are a length/status code, the 4 byte data source, and the string that was input.
- HOST is not a valid input source.

The returned parameters are a length/status code (similar to the numeric return value described above), the 4 byte data source, and the actual number that was input.

Host Command: INS

Host Parameters:		
	Keypad	KPD
	Scanner	SCAN
	All	AL
	DefaultValue	DEF
	ContinueOnSpecialKeys	SKC
	QuitOnKey	QKY
	ContinueOnNoDecode	NDC
	QuitOnNoDecode	NDQ
	CursorOn	CURE
	CursorOff	CURD
	SilentInput	INSL
	RefreshDisplayAfterInput	REFR
	FlushBuffer	DMP
	PreserveBuffer	SAV
	Minimum	MIN
	Maximum	MAX
	Timeout	TMOUT

Example

In this example, a string is input from any source (the default is “Test”) and quits if an arrow key is hit. If successful, the string is displayed.

```
Comment InputString Sample Program
String  S1*32


S1="Test"
If(InputString(S1, Keypad, DefaultValue=S1,
QuitOnKey=17, QuitOnKey=18) >= 0)
{
OutputString("String ="+S1)
}
```

InterpreterVersion

This function returns the interpreter version.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

InterpreterVersion

Return Value

An interpreter version string is returned.

Remote Host Command Compatability

Syntax is the same as above. The returned parameter is a string containing the interpreter version.
Host Command: VERIN

Example

In the following example, the interpreter version is written to the display.

```
Comment InterpreterVersion Sample Program
OutputString(InterpreterVersion)
```

KeyFlush

This function flushes any keys from the keypad buffer.

Syntax

KeyFlush

Return Value

None.

Remote Host Command Compatability

Syntax is the same as above. Returns 1 if the command was correct, and an error code otherwise (see Bad Response on page 3-6).
Host Command: KFL

Example

In this example, any keys in the keypad buffer that haven't been read are flushed, and GetKey only returns a key code when a key is pressed.

```
Comment KeyFlush Sample Program
Int     I1
KeyFlush
I1 = GetKey
```

KeyLook

KeyLook gets the scan code of the oldest keypress in the keypad buffer, or the key being pressed when the keypad buffer is empty. KeyLook always returns immediately. When the keypad buffer is used for input, the input character is not deleted from the keypad buffer. If you are watching for keys that might start an input sequence, so that InputString can be called to read the sequence, KeyLook leaves the first key in the keypad buffer for InputString to read. (If you were using GetKey, that key would be deleted.) Refer to Numeric Key Codes, page 8-6 for an illustration of the keypad and the number key codes assigned to each key and each FN-key sequence.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

KeyLook

Return Value

Returns the integer scan code of the key code read from the keypad buffer, or the key being pressed. If no key was read, 0 is returned. The following keys elicit the following returns:

Key	Returns	Function Key	Returns
1	1	1	65
2	2	2	66
3	3	3	67
4	4	4	68
5	5	5	69
6	6	6	70
7	7	7	71
8	8	8	72
9	9	9	73
0	10	0	74
*	11	down arrow	81
Enter	16	up arrow	82
down arrow	17		
up arrow	18		
backspace	19		

Note: In the case of a FN-key sequence, the FN key is pressed and released and then another key is pressed.

Remote Host Command Compatability

Syntax is same as above. Returns the scan code of the key that was read.
Host Command: KLK

Example

In this example, we'll look for the "1" key to be pressed. As soon as we see it, we'll go get a string from the keypad, which will include the "1" key as its first character.

```
Comment KeyLook Sample Program
Int     I1
String  S1*32

Do  {
        I1 = KeyLook
        If(I1<>1)
        {
        I1 = GetKey
        }
}  While (I1 <> 1)
InputString(S1, Source=Keypad, PreserveBuffer)
```

LeftJustify

This function returns a string that has been left justified or truncated according to n.

Syntax

LeftJustify(
[String=]*user-defined string*,
[Total=] n,
[[Fill=]'x']
)

Parameters

String=

A string expression that specifies the string to be left justified or truncated.

Total=

The integer constant n must be in the range of 1 to 3000 and specifies the total number of characters in the output. If n is less than the size of string, the string is truncated from the right. If n is greater than the size of string, it is padded with the specified fill character or with nulls.
Minimum value: 1
Maximum value: 3000

Fill=

Fill characters to be used if n is greater than the size of the string. Default: Null

Return Value

Returns the left justified or truncated string.

Remote Host Command Compatability

Not compatible.

Example

In this example, the first OutputString adds a zero fill character to the end of “abcdefg”. In the second OutputString, “abcdefg” is truncated to “abcde”.

```
Comment LeftJustify Sample Program
String  S1*32

S1 = "abcdefg"

OutputString(LeftJustify(S1, 8, ‘0’))
OutputString(LeftJustify(S1, 5, ‘ ‘))
```

Length

Gets the number of characters in the user-defined string expression.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

Length(
user-defined string expression
)

Parameters

user-defined string expression

The string expression whose length is to be determined.

Return Value

Integer number of characters in the string expression.

Remote Host Command Compatability

Not compatible.

Example

In this example, the “Size of S1 = 8” is written to the display.

```
Comment Length Sample Program
String  S1*32

S1="AbCdEfG"
OutputString("Size of S1 = "+ String(Length(S1)))
```

LoadImage

Loads a specified image to the display. Refer to Chapter 8 for representations of these images and their bitmap size.

Syntax

LoadImage(
[Image=]*user-defined numeric expression 1* | Antenna | ArrowDown | ArrowLeft | ArrowRight | ArrowUp | AssocInProg | BackLightOff | BackLightOn | BarLarge | BarMedium | BarSmall | BarXLarge | BarXSmall | BatteryHigh | BatteryLow | Border | BulletRound | BulletSquare | Busy | ButtonBlank | ButtonInverse | ButtonCancel | ButtonNo | ButtonOK | ButtonYes | Clock | Contrast | FaceSad | FaceSmiley | Failure | FunctionKey | LogoHHP | LogoIT3875 | Success | ThumbDown | ThumbUp | TruncationSign | Warning,
[x=]*user-defined numeric expression 2*,
[y=]*user-defined numeric expression 3*
)

Parameters

Image=

The image may be specified either by name or by an index value from 1 to 38. See "Bitmap Chart" on page 8-1 to view the bitmap images.
Minimum value: 1
Maximum value: 38

x=

A numeric expression defining the x coordinate on the display where the left edge of the bitmap image will appear.
Minimum value: 0
Maximum value: Number of display columns, minus 1

y=

A numeric expression defining the y coordinate on the display where the top edge of the bitmap image will appear.
Minimum value: 0
Maximum value: Number of display rows, minus 1

Return Value

Boolean true if the operation succeeded and the coordinates were valid, false otherwise.

Remote Host Command Compatability

Syntax is same as above, except that no parameter labels are allowed, and the numeric expressions for the X,Y coordinates of the image index (if used in place of the name) must be numeric constants. Returns 1 for true if the image loaded correctly, 0 otherwise.

Host Command: LDG

Host Parameters:

Antenna	G01
ArrowDown	G02
ArrowLeft	G03
ArrowRight	G04
ArrowUp	G05
AssocInProg	G06
BackLightOff	G07
BackLightOn	G08
BarLarge	G09
BarMedium	G10
BarSmall	G11
BarXLarge	G12
BarXSmall	G13
BatteryHigh	G14
BatteryLow	G15
Border	G16
BulletRound	G17
BulletSquare	G18
Busy	G19
ButtonBlank	G20
ButtonInverse	G21
ButtonCancel	G22
ButtonNo	G23
ButtonOK	G24
ButtonYes	G25
Clock	G26
Contrast	G27
FaceSad	G28
FaceSmiley	G29
Failure	G30
FunctionKey	G31
LogoHHP	G32
LogoIT3875	G33
Success	G34
ThumbDown	G35
ThumbUp	G36
TruncationSign	G37
Warning	G38

Example

In the example below, the Busy image is loaded to the display starting at (10,10), and the Antenna image is loaded to the display starting at (0,0).

```
Comment LoadImage Sample Program
LoadImage(Busy,  10, 10)
LoadImage(1,  0, 0)
```

LowerCase

This function converts all alphabetic characters in the user-defined string expression to lower case.

Syntax

LowerCase(
user-defined string expression
)

Return Value

Returns the converted string.

Remote Host Command Compatability

Not compatible.

Example

In this example, the OutputString displays “abcdefg.”

```
Comment LowerCase Sample Program
String  S1*32

S1 = "AbCdEfG"

OutputString(LowerCase(S1))
```

MenuCommand

This function processes the user-defined string expression as a menu command.

Syntax

MenuCommand(
user-defined string expression
)

Return Value

When MenuCommand is part of a Boolean expression, it returns true if a number is found, false otherwise. When MenuCommand is part of a string expression, it returns one of the following:

- A null string if the menu command didn't process properly
- The original command string if the menu command processed properly, but didn't produce a string result
- The string result from processing the menu command, if it processed properly and produced a string result. An example is a menu setting query command that returns its current value.

Remote Host Command Compatability

Syntax is the same as above except the string expression must be a string constant. The returned parameters are a status code and the string result from processing the menu setting command. The status code is 1 for true if the command executed correctly, 0 otherwise.
Host Command: MNU

Example

In this example, Codabar decoding is enabled.

```
Comment MenuCommand Sample Program
If(MenuCommand("CBRENA1"))
{
        OutputString("Codabar is enabled")
}
```

MoveDisplayCursorTo

Moves the cursor location to the specified X and Y coordinates.

Syntax

MoveDisplayCursorTo(
[x=]*user-defined numeric expression 2*,
[y=]*user-defined numeric expression 2*
)

Parameters

x=

A numeric expression defining the x coordinate on the display where the left edge of the cursor is to be placed.
Minimum value: 0
Maximum value: Number of display columns, minus 1

y=

A numeric expression defining the y coordinate on the display where the top edge of the cursor is to be placed.
Minimum value: 0
Maximum value: Number of display rows, minus 1

Return Value

Boolean true is returned if the command worked correctly, false otherwise.

Remote Host Command Compatability

Syntax is same as above, except that no parameter labels are allowed and the numeric expressions for the X,Y coordinates must be numeric constants. Returns 1 for true if the command worked correctly, 0 otherwise.
Host Command: CUR

Example

In this example, the cursor is moved to coordinates (10,10) on the display and “Display Test” is written there.

```
Comment MoveDisplayCursorTo Sample Program
MoveDisplayCursorTo(10,10)
OutputString("Display Test")
```

OutputString

This function writes the user-defined string expression to the display at the current cursor location.

Note: This function does not send data to the host PC; it only displays a string to the scanner LCD.

Syntax

OutputString(
[String=]*user-defined string expression*,
[[Video=]Normal | Inverse],
[[Font=]*user-defined numeric expression* | VeryClearSmall | VeryClearMedium | VeryClearLarge],
[SelectForHorizontalScrolling],
[LeftJustify | RightJustify]
)

Parameters

String=

The string expression to be output to the display.

Video=

Normal	Characters are displayed as black on white.
Inverse	Characters are displayed as white on black.

Default: Normal

Font=

An integer or a font name may be specified.
0 or VeryClearSmall
1 or VeryClearMedium
2 or VeryClearLarge

Minimum value: 0
Maximum value: 2
Default: Active font

SelectForHorizontalScrolling

The string being displayed may be scrolled horizontally. Horizontal scrolling permits the user to see a string that is too long to fit on the display. Only one string may be selected for horizontal scrolling on the display at any given time. The last string written to the display with this attribute is the one that scrolls. When horizontal scrolling is selected, this function suspends until a non-scroll key is hit.
Default: No SelectForHorizontalScrolling

LeftJustify | Right Justify

You may specify left or right justification for the string when it is too long to fit on the display. Default: LeftJustify

Return Value

Returns true if the operation succeeded, false otherwise.

Remote Host Command Compatability

Syntax is same as above, except that no parameter labels are allowed and the string expression must be a string constant. Returns 1 for true if the command worked correctly, 0 otherwise.
Host Command: OUTS

Host Parameters:		
	Normal	NRM
	Inverse	INV
	VeryClearSmall	VCS or 0
	VeryClearMedium	VCM or 1
	VeryClearLarge	VCL or 2
	SelectForHorizontalScrolling	SCLH
	LeftJustify	LJ
	RightJustify	RJ

Example

In this example, “Very Long Test String” is displayed in inverse mode, using a large font, and selected for horizontal scrolling.

```
Comment OutputString Sample Program
OutputString("Test String", Inverse,
Font=VeryClearLarge, SelectForHorizontalScrolling)
```

RadioStrength

RadioStrength returns the radio signal strength on a scale of 0 (no signal) to 100 (excellent signal).

Note: The IT3875 returns 0 if not connected or not within range of the radio base or returns 100 if connected and communicating to the base radio.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

RadioStrength

Return Value

Returns the integer radio signal strength.

Remote Host Command Compatability

Syntax is same as above.
Host Command: RADSIG

Example

In this example, the radio signal strength is written to the display.

```
Comment RadioStrength Sample Program
OutputString("Radio signal strength =
"+string(RadioStrength))
```

ResetTimer0

This command resets the system timer to zero. To find out the number of milliseconds that have elapsed since the timer was reset, use GetTimer0 (see page 4-27). After Timer0 counts up to 65,535, it starts again from zero.

Syntax

ResetTimer0

Return Value

None.

Remote Host Command Compatability

Not compatible.

Example

In the example below, the input operation is timed and the time is displayed. See "GetKey" on page 4-25 for more details.

```
Comment ResetTimer0 Sample Program
Int     I1
String  S1*32


ResetTimer0
InputString(S1)
I1=GetTimer0
OutputString("You took "+string(I1)+" milliseconds to
input data!")
```

RightJustify

This function returns a string that has been right justified or truncated according to n.

Syntax

RightJustify(
[String=]*user-defined string*,
[Total=]n,
[[Fill=]'x']
)

Parameters

String=

A string expression that specifies the string to be right justified or truncated.

Total=

The integer constant n must be in the range of 1 to 3000 and specifies the total number of characters in the output. If n is less than the size of string, the string is truncated from the left. If n is greater than the size of string, it is padded with the specified fill character or with nulls.
Minimum value: 1
Maximum value: 3000

Fill=

Fill characters to be used if n is greater than the size of the string.
Default: Null

Return Value

Returns the right justified or truncated string.

Remote Host Command Compatability

Not compatible.

Example

In this example, the first OutputString adds a zero fill character to the beginning of “abcdefg”. In the second OutputString, “abcdefg” is truncated to “cdefg.”

```
Comment RightJustify Sample Program
String  S1*32

S1 = "abcdefg"

OutputString(RightJustify(S1, 8, ‘0’))
OutputString(RightJustify(S1, 5, ‘ ‘))
```

ScannerID

This function returns the scanner ID. The scanner ID is always four characters.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

ScannerID

Return Value

The scanner ID string is returned.

Remote Host Command Compatability

Syntax is the same as above. The returned parameter is a string containing the scanner ID.
Host Command: SCNID

Example

In this example, the scanner ID is written to the display.

```
Comment ScannerID Sample Program
OutputString(ScannerID)
```

ScrollDisplay

Scrolls the display up or down by the number of pixels specified in the numeric expression. A negative count scrolls up; a positive count scrolls down.

Syntax

ScrollDisplay (
user-defined numeric expression
)

Return Value

Boolean true is returned if this operation works; false otherwise.

Remote Host Command Compatability

Syntax: ScrollDisplay(# of lines), where # of lines to scroll is a numeric constant. Returns 1 for true if the command was correct and an error code otherwise.
Host Command: SCRL

Example 1

In the example below, the display is scrolled up 10 pixels.

```
Comment ScrollDisplay Sample Program
ScrollDisplay (-10)
```

Example 2

In the example below, the display is scrolled down the height of the active font.

```
Comment ScrollDisplay Sample Program
ScrollDisplay (CharacterHeight)
```

Send

This command sends a string to the host. One of the uses of Send is to report bar code data, which sometimes needs to be wrapped with prefix text and suffix text.

Syntax

```
Send(
  [String=]user-defined string expression 1,
  [[Prefix=]DefaultPrefix | PrefixAlways(user-defined string expression)],
  [[Suffix=]DefaultSuffix | SuffixAlways(user-defined string expression))],
  [[DataSource=]user-defined string expression 2],
  [UseIndicatorFeedback | DoNotUseIndicatorFeedback]
  )
```

Note: If you are using IQ App Loader v1.1.1 or IQ App Designer v1.1.2 (or earlier) and you include the SuffixAlways parameter, you must use two closing parentheses after the user-defined string expression. For example, Send(LastDecodedMessage, SuffixAlways("\0d\0a")))

Parameters

String=

A string expression that is sent to the host port.

Prefix=

DefaultPrefix	A globally defined prefix is applied (see DefaultPrefix on page 4-12).
ImmediatePrefixAlways	The string expression is written as a prefix under all conditions.
ImmediatePrefixWhenDataFound	The string expression is written as a prefix when the Send string expression is not null.
ImmediatePrefixWhenDataNotFound	The string expression is written as a prefix when the Send string expression is null.

Default: No Prefix

Suffix=

DefaultSuffix	A globally defined suffix is applied (see DefaultSuffix on page 4-12).
ImmediateSuffixAlways	The string expression is written as a suffix under all conditions.
ImmediateSuffixWhenDataFound	The string expression is written as a suffix when the Send string expression is not null.
ImmediateSuffixWhenDataNotFound	The string expression is written as a suffix when the Send string expression is null.

Default: No Suffix

DataSource=

The data source identifier may be stored to this string variable. The HHP code ID in the first byte, followed by the 3 letter AIM symbology ID code, indicates the device from which input was received, and information about the type of symbol read (if applicable). A value of "9]Z9" indicates the string sent to the host was not generated by an input device. For additional information, please refer to the IT3870/ 75/5770 System Manual. Default: 9]Z9

UseIndicatorFeedback / DoNotUseIndicatorFeedback

UseIndicatorFeedback is used when you want to perform good or bad LED/beep sequences.
DoNotUseIndicatorFeedback is used when you do not want to perform good or bad LED/beep sequences.
Default: UseIndicatorFeedback

Return Value

Boolean true is returned if the command worked correctly, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

The first Send sends the string "abcd" to the host with a default data source of "9]Z9." The second Send sends the string "abcd" with the four-character Scanner ID included as a prefix, and a data source of "k]Z0" to indicate keypad input. The third Send sends the message in pre-defined variable LastDecodedMessage using a data source that is stored in user variable LastDataSource.

```
Comment Send Sample Program
string LastDataSource*4

Send("abcd")

Send(string="abcd", PrefixAlways(ScannerID), "k]Z0")

Send(LastDecodedMessage, DataSource=LastDataSource)
```

SetFont

Sets the active font according to the font name and size specified.

Control characters (00-1F Hex) take up two times the character width, in pixels. All other characters take one character width. Refer to the table below for the font size in pixels.

Font	***Size in Pixels***
VeryClearSmall	6w x 8h
VeryClearMedium	8w x 11h
VeryClearLarge	12w x 16h

Syntax

SetFont(
VeryClearSmall | VeryClearMedium | VeryClearLarge
)

Parameters

VeryClearSmall | VeryClearMedium | VeryClearLarge

An integer or a font name may be specified.
0 or VeryClearSmall
1 or VeryClearMedium
2 or VeryClearLarge

Minimum value: 0
Maximum value: 2
Default: The active font

Return Value

None.

Remote Host Command Compatability

Syntax is same as above. Returns 1 for true if the command worked correctly, 0 otherwise.
Host Command: SFNT

Host Parameters:		
	VeryClearSmall	VCS or 0
	VeryClearMedium	VCM or 1
	VeryClearLarge	VCL or 2

Example

In the example below, the active font display is VeryClear with a large character height.

```
Comment SetFont Sample Program
SetFont(VeryClearLarge)
```

SetLength

Set the length of the string variable to the value in the numeric expression.

Syntax

SetLength (
user-defined string variable,
user-defined numeric expression
)

Return Value

Boolean true is returned if the command worked correctly, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

In the example below, S1 was originally assigned a 4 character string, but the length was changed to 2, so Send only reports the first two characters. After reassigning the original string to S1 and changing the length to 6, Send reports 6 characters, where the last two characters are an undetermined value, since they haven't been explicitly set.

```
Comment SetLength Sample Program
String  S1*32
S1="abcd"
SetLength(S1,2)
Send(S1)
S1="abcd"
SetLength(S1,6)
Send(S1)
```

StringComparisonsAreCaseInsensitiveByDefault

String comparisons are defaulted to case insensitive.

Syntax

StringComparisonsAreCaseInsensitiveByDefault

Return Value

None.

Remote Host Command Compatability

Not compatible.

Example

In this example, “Strings compare” is written to the display.

```
Comment StringComparisonsAreCaseInsensitiveByDefault
Sample Program
String  S1*32


S1="abcd"
StringComparisonsAreCaseInsensitiveByDefault
If(S1 = ""ABCD")
{
        OutputString("Strings compare")
}
else
{
        OutputString("Strings don’t compare")
}
```

StringComparisonsAreCaseSensitiveByDefault

String comparisons are defaulted to case sensitive.

Syntax

StringComparisonsAreCaseSensitiveByDefault

Return Value

None.

Remote Host Command Compatability

Not compatible.

Example

In this example, “Strings don’t compare” is written to the display.

```
Comment StringComparisonsAreCaseSensitiveByDefault
Sample Program
String  S1*32


S1="abcd"
StringComparisonsAreCaseSensitiveByDefault
If(S1 = "ABCD")
{
        OutputString("Strings compare")
}
else
{
        OutputString("Strings don’t compare")
}
```

TemporarilyEnableBacklight

This function temporarily enables backlighting on the display for a specified period of time. When the time has expired, the backlight is turned off, even if had previously been on.

TemporarilyEnableBacklight(
[Time=]*user-defined numeric expression*
)

Parameters

Time=

Enter the length of time (in milliseconds) you want the backlighting to remain on. Control returns immediately to the program after turning on backlighting. Backlighting is turned off automatically in the background when the timer expires.
Minimum value: 0
Maximum value: 64000 (64 seconds)

Return Value

None.

Remote Host Command Compatability

Syntax is same as above, except that no parameter labels are allowed and the numeric expression must be a numeric constant. Returns 1 for true if the command worked correctly, 0 otherwise.
Host Command: TBLE

Example

In the example below, backlighting is turned on for 50 milliseconds.

```
Comment TemporarilyEnableBacklight Sample Program
TemporarilyEnableBacklight(50)
```

TriggerHit

This function determines if the trigger is being pressed.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

TriggerHit

Return Value

Boolean true if the trigger is depressed, false otherwise.

Remote Host Command Compatability

Syntax: TriggerHit. Returns 1 for true if the trigger is depressed, 0 otherwise.
Host Command: TRG

Example

In this example, a message is displayed if the trigger is depressed.

```
Comment TriggerHit Sample Program
If(TriggerHit)
{
        OutputString("Trigger is depressed")
}
```

UpperCase

This function converts all alphabetic characters in the string expression to upper case.

Syntax

UpperCase(
user-defined string expression
)

Return Value

Returns the converted string.

Remote Host Command Compatability

Not compatible.

Example

In this example, the OutputString displays “ABCDEFG.”

```
Comment UpperCase Sample Program
String  S1*32

S1 = "AbCdEfGc

OutputString(UpperCase(S1))
```

Wait

This command idles the unit for a specified time. Enter the length of time (in milliseconds) you want the unit to idle. The range is 0 to 64,000 (64 seconds). The accuracy of the timing may be plus or minus several milliseconds.

Syntax

Wait(
user-defined numeric expression
)

Return Value

None.

Remote Host Command Compatability

Not compatible.

Example

In this example, the string “Before Wait” is written to the display, then 575 milliseconds later the string “AfterWait” is written to the display.

```
Comment Wait Sample Program
OutputString("Before Wait")
Wait(575)
OutputString("After Wait")
```

WaitForCommand

This command waits for a single host command to be issued via the host communication port. This function does not return control to the program until a command has been received.

Syntax

WaitForCommand(
[[HostRoutines=]NONE | HOST | *(list)*]
)

Parameters

NONE | HOST | string expressions

NONE	No routines can be executed by the host while in host command mode.
HOST	Only host routines can be executed by the host.
list	String expressions specifying the names of routines that can be executed by the host. For example, WaitForCommand(HostRoutines=(“AB”,”CD”)) indicates that the host can execute host routines AB and CD.

Default: HOST

Return Value

If a host routine is executed via the host command Execute, the routine name is returned as a string. Otherwise, a null string is returned.

Remote Host Command Compatability

Not compatible.

Example

In the example below, the first OutputString is executed, the VersaCode processor waits for a host VersaCode command to be received and processed, then the second OutputString is executed.

```
Comment WaitForCommand Sample Program
OutputString("Waiting for VersaCode command")
WaitForCommand
OutputString("A VersaCode command was executed")
```

WaitForKey

WaitForKey gets the scan code of the oldest keypress in the keypad buffer and removes it from the buffer. If the keypad buffer is empty, WaitForKey does not return until either a key appears in the keypad buffer, or the optional timeout expires. Optionally, the keypad buffer may be flushed on entry to the routine. Refer to Numeric Key Codes, page 8-6 for an illustration of the keypad and the number key codes assigned to each key and each FN-key sequence.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

```
WaitForKey (
  [buffer control,]
  [timeout=user-defined numeric expression]
  )
```

Parameters

buffer control

FlushBuffer — When the buffer is discarded, WaitForKey waits for a new key entry in the keypad buffer.

PreserveBuffer — When the buffer is preserved and there are one or more entries in the keypad buffer, the first entry in the buffer is returned.

Default: FlushBuffer

timeout

A time out (in milliseconds) may be specified to indicate how long to wait for a new key before terminating the WaitForKey operation.

Minimum value: 0

Maximum value: 64000 (64 seconds)

Default: No time out

Return Value

Returns the integer scan code of the key code read from the keypad buffer, or the key being pressed. If no key was read, 0 is returned. The following keys elicit the following returns:

Key	Returns	Function Key	Returns
1	1	1	65
2	2	2	66
3	3	3	67
4	4	4	68
5	5	5	69
6	6	6	70
7	7	7	71
8	8	8	72
9	9	9	73
0	10	0	74
*	11	down arrow	81
Enter	16	up arrow	82
down arrow	17		
up arrow	18		
backspace	19		

Note: In the case of a FN-key sequence, the FN key is pressed and released and then another key is pressed.

Remote Host Command Compatability

Syntax is the same as above. Returns the scan code of the key that was read.

Host Command: WAITKY

Host Parameters:		
	FlushBuffer	DMP
	Preserve Buffer	SAV
	Timeout	TMOUT

Example 1

In this example, the scan code of the next key in the keypad buffer is written to the display.

```
Comment WaitForKey Sample Program
OutputString("Key read = "+string(WaitForKey))
```

Example 2

In this example, we'll look for the "1" key to be pressed. As soon as we see it, we'll go get a string from the keypad, which will not include the "1" key.

```
Comment WaitForKey Sample Program
Int     I1
String  S1*32

Do  {
        I1 = WaitForKey
        }
}  While (I1 <> 1)
InputString(S1, Source=Keypad)
```

WaitForMultipleCommands

This function waits for multiple host commands to be issued via the host communication port. This function returns control to the program when the "End" host command is sent. Refer to Host Command Examples, page 3-7.

Syntax

WaitForMultipleCommands(
[[HostRoutines=]NONE | HOST | *(list)*]
)

Parameters

NONE | HOST | string expressions

NONE	No routines can be executed by the host while in host command mode.
HOST	Only host routines can be executed by the host.
list	String expressions specifying the names of routines that can be executed by the host. For example, WaitForMultipleCommands(HostRoutines=("AB","CD")) indicates that the host can execute host routines AB and CD.

Default: HOST

Return Value

If a host routine is executed via the host command Execute, the name of the last routine executed is returned as a string. Otherwise, a null string is returned.

Remote Host Command Compatability

Not compatible.

Example

In the example below, the first OutputString is executed, the VersaCode processor waits for multiple host VersaCode command to be received and processed until an “End” host VersaCode command is received, then the second OutputString is executed.

```
Comment WaitForMultipleCommands Sample Program
OutputString("Waiting for VersaCode commands")
WaitForMultipleCommands
OutputString("Host sent an END command")
```

WildcardCompare

This function compares two strings, using wildcards “*” and “?” in the first string.

Syntax

WildcardCompare(
user-defined string expression 1,
user-defined string expression 2
)

Parameters

user-defined string 1

Wildcard characters “*”, “#”, and “?” may be used in user-defined string 1. A “*” in user-defined string 1 indicates that zero or more characters in user-defined string 2 can be ignored. A “?” in user-defined string 1 indicates that the next character in user-defined string 2 can be any character. A “#” in user-defined string 1 indicates that the next character in user-defined string 2 must be numeric.

If you are searching for a question mark (?), pound sign (#), or asterisk (*) character in a string, enter the search beginning with a caret (^). For example, to search for "A?B*C," enter the search string "A^?B^*C." If you are searching for a caret mark, use two marks (^^). For example, to search for "A^B," enter the search string "A^^B."

user-defined string 2

The user-defined string to which user-defined string 1 is compared.

Using “abcdefg” as user-defined string 2, the following examples produce the accompanying results:
a user-defined string 1 value of “a*ef*” produces a match

a user-defined string 1 value of “a?cdef?” produces a match
a user-defined string 1 value of “a?bcdefg?” does not produce a match (there must be exactly one character corresponding to the ‘?’)
The default setting for case sensitivity is used in this comparison.

Return Value

Boolean true if the strings compare, false otherwise.

Remote Host Command Compatability

Not compatible.

Example

In this example, the compares are case sensitive by default, so the first and fourth OutputString calls are made.

```
Comment WildcardCompare Sample Program
String  S1*32

S1="abcdefg"
If(WildcardCompare("a*ef*",S1))
{
OutputString("S1 equals a*ef*")
}
else
{
OutputString("S1 doesn’t equal a*ef*")
}
If(WildcardCompare("a*ef??",S1))
{
OutputString("S1 equals a*ef??")
}
else
{
OutputString("S1 doesn’t equal a*ef??")
}
```

Workgroup

Workgroup returns the current workgroup number. Workgroups are features of the cordless system. Refer to the IT3870/75/5700 Cordless System Manual for additional information.

Note: The return value must be assigned to a variable or used in an expression.

Syntax

Workgroup

Return Value

Returns the current integer workgroup number.

Remote Host Command Compatability

Syntax is the same as above.
Host Command: GRP

Example

In this example, the workgroup number is written to the display.

```
Comment Workgroup Sample Program
OutputString("Current workgroup =
"+string(Workgroup))
```

5 Data Formatting Functions

Introduction

VersaCode provides commands to implement the basic HHP data formatting capability available in all scanners. While these commands should not be needed, as VersaCode provides more powerful ways of achieving what these commands are doing, we still offer them for backward compatibility.

All data formatting operations use the pre-defined variable, **LastDecodedMessage**, as a data source. It is assumed, therefore, that **LastDecodedMessage** has been loaded with data to be edited via InputString (see page 4-36) or some other means.

The data formatting operations that update the edit cursor have been preserved as well. As a VersaCode programmer, you can access the edit cursor via the pre-defined variable EditCursor. At the start of the VersaCode program, EditCursor is set to zero to point to the beginning of the data in **LastDecodedMessage**.

CompareNextCharacterTo

Compares the character at the EditCursor in the data being edited to x.

Syntax

```
CompareNextCharacterTo(
  'x'
  )
```

Parameters

x

The string constant that specifies the character to be compared.

Return Value

Returns true for a match, and false otherwise.

IsNextCharacterNonNumeric

Determines if the character at the EditCursor is non-numeric.

Syntax

```
IsNextCharacterNonNumeric
```

Return Value

Returns true if the character at the EditCursor in the data being edited is non-numeric, false otherwise.

IsNextCharacterNumeric

Determines if the character at the EditCursor is numeric.

Syntax

IsNextCharacterNumeric

Return Value

Return true if the character at the EditCursor in the data being edited is numeric, false otherwise.

MoveCursorAheadBy

Moves the EditCursor ahead by a specified number.

Syntax

MoveCursorAheadBy(
n
)

Parameters

n

n is a numeric constant that specifies how far to move the EditCursor.
Minimum value: 1
Maximum value: 99

Return Value

None.

MoveCursorAheadToChar

Moves the EditCursor ahead to the first character equal to x.

Syntax

MoveCursorAheadToChar(
'*x*'
)

Parameters

x

The string constant that specifies the character to which the EditCursor will be moved.

Return Value

None.

MoveCursorAheadToNonChar

Moves the EditCursor ahead to the first character not equal to x.

Syntax

MoveCursorAheadToNonChar(
'*x*'
)

Parameters

x

The string constant that specifies the character to which the Edit Cursor will move beyond.

Return Value

None.

MoveCursorBackBy

Moves the EditCursor back by a specified number.

Syntax

MoveCursorBackBy(
n
)

Parameters

n

n is a constant integer that specifies how far back to move the EditCursor.
Minimum value: 1
Maximum value: 99

Return Value

None.

MoveCursorBackToChar

Moves the EditCursor back to the first character equal to x.

Syntax

MoveCursorBackToChar(
'*x*'
)

Parameters

x

The string constant that specifies the character to which the EditCursor will be moved.

Return Value

None.

MoveCursorBackToNonChar

Moves the EditCursor back to the first character not equal to x.

Syntax

MoveCursorBackToNonChar(
'*x*'
)

Parameters

x

The string constant that specifies the character to which the Edit Cursor will move beyond.

Return Value

None.

MoveCursorToBeginning

Moves the EditCursor to the beginning of the data being edited.

Syntax

MoveCursorToBeginning

Return Value

None.

MoveCursorToEnd

Moves the EditCursor to the end of the data being edited.

Syntax

MoveCursorToEnd

Return Value

None.

ReplaceCharacters

Substitutes replacement for characters in the data being edited as it is sent to the host communication port.

Syntax

ReplaceCharacters(
‘*x1a*’ with ‘*x1b*’,
‘*x2a*’ with ‘*x2b*’,
… ‘*xna*’ with ‘*xnb*’
)

Parameters

‘x1a’ with ‘x1b’ ... ‘xna’ with ‘xnb’

Using string constants, specify the characters to be replaced in the data with the characters that will replace them. When more than one replacement pair is specified, separate the pairs with commas. Up to 15 replacement pairs can be specified.
Minimum value: 1 pair
Maximum value: 15 pairs

Return Value

None.

SuppressCharacters

Suppresses the output of the specified characters in the data being edited.

Syntax

SuppressCharacters(
‘x1’,
‘x2’,
… ‘xn’
)

Parameters

x1 ... xn

The string constant that specifies the character you wish to suppress in the output. When more than one suppression character is specified, separate the characters with commas.
Minimum value: 1
Maximum value: 15

Return Value

None.

TurnOffCharacterReplacement

Turns off character replacement in the data being edited.

Syntax

TurnOffCharacterReplacement

Return Value

None.

TurnOffCharacterSuppression

Turns off character suppression in the data being edited.

Syntax

TurnOffCharacterSuppression

Return Value

None.

WriteAllButLastCharactersStartingAtCursor

Sends all characters from the EditCursor, minus a specified number of characters from the end of the data being edited.

Syntax

```
WriteAllButLastCharactersStartingAtCursor(
  n
  )
```

Parameters

n

n is a numeric constant that specifies the number of characters to be omitted from the end of the data being edited.
Minimum value: 1
Maximum value: 99

Return Value

None.

WriteCharactersFromCursor

Sends a specified number of characters from the EditCursor, tacking an optional character at the end.

Syntax

```
WriteCharactersFromCursor (
  n,
  [[End=‘x’]
  )
```

Parameters

n

n is a numeric constant that specifies how many characters from the EditCursor will be written.
Minimum value: 1
Maximum value: 99

End=

A string constant that specifies a character to be sent to the host after the edit buffer data.

Return Value

None.

WriteCharactersFromCursorToChar

Sends characters from EditCursor to the first occurrence of a specified character, tacking an optional character at the end.

Syntax

```
WriteCharactersFromCursorToChar(
  ‘x’,
  [[End=‘y’]
  )
```

Parameters

x

A string constant that specifies the character to stop at when sending edit buffer data to the host.

End=

A string constant that specifies a character to be sent to the host after the edit buffer data.

Return Value

None.

WriteCharactersFromCursorToEnd

Sends all characters from the EditCursor to the end of the data being edited, tacking an optional character at the end.

Syntax

```
WriteCharactersFromCursorToEnd(
  [[End=‘x’]
  )
```

Parameters

End=

A string constant that specifies a character to be sent to the host after the edit buffer data.

Return Value

None.

WriteCharMultipleTimes

Sends a character to the host port a specified number of times

Syntax

```
WriteCharMultipleTimes(
  'x',
  n times
  )
```

Parameters

x

A string constant that specifies the character to be sent to the host port.

n

n is a numeric constant that specifies the number of times character x is sent to the host port.
Minimum value: 1
Maximum value: 99

Return Value

None.

Sample Scripts

This section includes representations of factory scripts for you to use as reference as you create custom scripts. The actual scripts are included on the VersaCode CD-ROM that accompanied this manual or from the HHP website.

No Script (IT3870 Mode)

This script disables the LCD screen and disables keypad entry. Normal scanning and LED and beeper functions still work.

```
REM ************************************************************************
REM **
REM **  Script Name:    3870Mode.vc     (3870 mode default script for 3875)
REM **
REM **  Description:
REM **      This program causes the 3875 to function like a 3870,
REM **      implementing a loop where we continuously try to read bar code
REM **      data and send it to the host.  The display and keypad are
REM **      inoperative, although the function key may still be used to get
REM **      into the system menu.
REM **
REM ** (c) Copyright 2002 Hand Held Products, Inc.
REM **
REM ************************************************************************
```

```
REM ************************************************************************
REM **
REM **  Revision History
REM **      02/19/02 - Initial release.
REM **
REM **      Source Control Information
REM **          $RCSfile: 3870mode.vc $
REM **          $Revision: 1.4 $
REM **          $Author: HolzhauD $
REM **          $Date: 2002/02/21 21:04:02 $
REM **
REM ************************************************************************



REM ** Predefined VersaCode variables used in this script

REM **      LastDecodedMessage - Contains barcode data returned by InputString



REM ** Variable declarations (note all variables are global in scope)
```

```
    string  DataSourceInformation*8         REM ** Storage for the 4 byte data source
                                            REM **  returned by InputString.



REM ************************************************************************
REM **
REM **  Routine Name:   Main Program
REM **
REM **  Description:
REM **      Clear the display, then continuously look for a decoded symbol
REM **      and send the result to the host.
REM **
REM ************************************************************************
program
{
REM ** Clear the display
    DisplayCursorOff
    EraseDisplay


REM ** Continuously read and report symbol data.  While reading, ignore all input
```

```
REM ** sources but the scanner, and make sure that the display remains blank by
REM ** turning the cursor off and not displaying input text (SilentInput).  In
REM ** order to ensure that data sent to the host is properly formatted by the
REM ** data formatter, pass the data source from the InputString operation on to
REM ** the Send command.
    forever
    {
    REM ** If backlight was turned on via the function key, turn it back off.
        Displaybacklightoff

        InputString(LastDecodedMessage, Source=Scanner, cursoroff, SilentInput,
DataSource=DataSourceInformation)

        Send(LastDecodedMessage, DataSource=DataSourceInformation)
    }
}
end
```

History Script

This script keeps the history of the past ten scans on the display. After you scan a bar code or enter keypad data and the information is successfully communicated to the base, the bar code information appears on the top of the display. Data from the most recent scan appears at the top of the display while older data moves down one line at a time. The arrow keys may be used to scroll up and down through the scan history. The FN and arrows can be used to scroll left and right if the data is wider than the display.

```
REM ************************************************************************
REM **
REM **  Script Name:    History.vc      (Data history display script for 3875)
REM **
REM **  Description:
REM **      This program will accept either keypad input terminated with an
REM **      ENTER key or barcode data.  As data is read, it is sent to the
REM **      host.  It is also preserved in a history buffer of up to 10 entries.
REM **      The most recent data entries appear on the display with the newest
REM **      entry at the top.  The user may scroll through these entries via
REM **      the up and down arrow keys.  Each time the user presses the up or
REM **      down arrow, the next entry in the buffer is highlighted.  If the
REM **      highlighted entry extends beyond the right edge of the screen, the
REM **      user may horizontally scroll the entry via function up or down
REM **      arrow.
REM **
REM **      In order to conserve on memory, all entries but the most recent
REM **      entry in the history buffer are truncated to the first 64 characters
REM **      of data.  If an entry is less than or equal to 64 characters, it
```

```
REM **      is left alone.  To implement this, we actually maintain two buffers
REM **      where the first buffer stores the most recent entry in its' entirety
REM **      and the second buffer stores the remaining nine entries up to 64
REM **      characters for each entry.  When an entry is truncated, the 64th
REM **      character will be set to ASCII hex 81.
REM **
REM **      Note that when we need to compute the number of text lines on the
REM **      display, we do it by computing:
REM **              ((DisplayHeight + 1) / CharacterHeight)
REM **
REM ** (c) Copyright 2002 Hand Held Products, Inc.
REM **
REM ************************************************************************

REM ************************************************************************
REM **
REM **  Revision History
REM **      02/19/02 - Initial release.
REM **
REM **      Source Control Information
REM **          $RCSfile: history.vc $
REM **          $Revision: 1.7 $
REM **          $Author: HolzhauD $
REM **          $Date: 2002/02/21 20:58:36 $
```

```
REM **
REM ************************************************************************



REM ** Predefined VersaCode variables used in this script

REM **      LastDecodedMessage - Contains barcode data returned by InputString



REM ** Variable declarations (note all variables are global in scope)

    STRING  DataSourceInformation*8             REM ** Storage for the 4 byte data source
                                                REM **  returned by InputString.

    INT     NumberOfEntriesUsedInHistoryBuffer  REM ** Keep track of the number of entries in
                                                REM **  the history buffer that are currently
                                                REM **  in use.  Once the history buffer is
                                                REM **  full, this variable should remain set
                                                REM **  to 10, but until it's full, we use it
                                                REM **  to determine how many history buffer
                                                REM **  entries need to be redisplayed when we
```

```
                                        REM **  scroll.

Int     LengthOfInput                   REM ** Number of bytes in input data.


String  HistoryBufferEntryOne*2710      REM ** Storage for the most recent entry in the
                                        REM **  history buffer, preserving the entire
                                        REM **  entry.
String  HistoryBufferEntriesTwoToTen*64[9] REM ** Storage for the remaining nine entries in
                                        REM **  the history buffer, preserving up to 64
                                        REM **  characters of each entry.

Int     i                               REM ** Index variable for looping.

Int     LastKeyPressed                  REM ** Keypad scan code of the last key pressed.
                                        REM **  Used by ScrollThroughHistoryBuffer when
                                        REM **  scrolling through the history.

Int     HighlightedHistoryBufferEntry   REM ** Points to the entry in the history buffer
                                        REM **  that is to be highlighted for horizontal
                                        REM **  scrolling.  Values are 0 for current input
                                        REM **  line or 1 (most recent entry in history
                                        REM **  buffer) to 10 (least recent entry in
                                        REM **  history buffer)
```

```
    Int     HistoryBufferEntryAtScreenTop       REM ** Points to the entry in the history buffer
                                                REM **  that should appear at the top of the
                                                REM **  display.  Values are 0 for current input
                                                REM **  line or 1 (most recent entry in history
                                                REM **  buffer) to 10 (least recent entry in
                                                REM **  history buffer)

    Int     HistoryBufferEntryAtScreenBottom    REM ** Points to the entry in the history buffer
                                                REM **  that should appear at the bottom of the
                                                REM **  display.  Values are 0 for current input
                                                REM **  line or 1 (most recent entry in history
                                                REM **  buffer) to 10 (least recent entry in
                                                REM **  history buffer)

    Boolean ArrowKeyPressed                     REM ** Indicate whether or not the user pressed an
                                                REM **  arrow key while we're scrolling through
                                                REM **  the history buffer.

    Int     NextEntryToRedisplay                REM ** Next entry in the history buffer to
                                                REM **  redisplay.  Used in
ResdisplayBeginningEntriesOfHistoryBuffer.

    Int     NumberOfEntriesToRedisplay          REM ** Total number of history buffer entries to
                                                REM **  redisplay.  Used in
```

```
ResdisplayBeginningEntriesOfHistoryBuffer.




REM ************************************************************************
REM **
REM **  Routine Name:   UnhighlightCurrentEntry
REM **
REM **  Description:
REM **      Resdisplay the current highlighted entry without background highlighting on.
REM **
REM ************************************************************************
routine UnhighlightCurrentEntry
{
REM ** Unhighlight the currently highlighted history buffer entry if it falls within the range
REM ** of entries on the screen and it is a valid history buffer entry number.  Note that if
REM ** the currently highlighted entry is zero, which points to the next input line, then
```

```
REM ** no action will be taken.
    if((HighlightedHistoryBufferEntry >= HistoryBufferEntryAtScreenTop) AND
            (HighlightedHistoryBufferEntry <= HistoryBufferEntryAtScreenBottom) AND
            (HighlightedHistoryBufferEntry > 0) AND
            (HighlightedHistoryBufferEntry <= NumberOfEntriesUsedInHistoryBuffer))
    {
    REM ** Erase the display line where the highlighted entry will redisplayed without
    REM ** highlighting.
        MoveDisplayCursorTo(0, ((HighlightedHistoryBufferEntry - HistoryBufferEntryAtScreenTop) *
CharacterHeight))
        EraseLine

    REM ** Get the entry to be unhighlighted from the appropriate history buffer and display
    REM ** it.
        if(HighlightedHistoryBufferEntry = 1)
        {
            OutputString(HistoryBufferEntryOne, Normal, LeftJustify)
        }
        else
        {
            OutputString(HistoryBufferEntriesTwoToTen[HighlightedHistoryBufferEntry-2],
                                                Normal, LeftJustify)
        }
    }
}
```

```
end     REM **  UnhighlightCurrentEntry



REM ************************************************************************
REM **
REM **  Routine Name:   HighlightCurrentEntry
REM **
REM **  Description:
REM **      Resdisplay the current highlighted entry with background highlighting on.
REM **
REM ************************************************************************
routine HighlightCurrentEntry
{
REM ** Display the currently highlighted history buffer entry if it falls within the range
REM ** of entries on the screen and it is a valid history buffer entry number.  Note that if
REM ** the currently highlighted entry is zero, which points to the next input line, then
REM ** no action will be taken.
    if((HighlightedHistoryBufferEntry >= HistoryBufferEntryAtScreenTop) AND
            (HighlightedHistoryBufferEntry <= HistoryBufferEntryAtScreenBottom) AND
            (HighlightedHistoryBufferEntry > 0) AND
            (HighlightedHistoryBufferEntry <= NumberOfEntriesUsedInHistoryBuffer))
    {
```

```
    REM ** Erase the display line where the highlighted entry will appear.
        MoveDisplayCursorTo(0, ((HighlightedHistoryBufferEntry - HistoryBufferEntryAtScreenTop) *
CharacterHeight))
        EraseLine

    REM ** Get the highlighted entry from the appropriate history buffer and display it.  If
    REM ** the entry doesn't fit in the display, the user will be allowed to scroll it
    REM ** horizontally via the function up and down arrows.
        if(HighlightedHistoryBufferEntry = 1)
        {
            OutputString(HistoryBufferEntryOne, Inverse,
                                        SelectForHorizontalScrolling, LeftJustify)
        }
        else
        {
            OutputString(HistoryBufferEntriesTwoToTen[HighlightedHistoryBufferEntry-2],
                                    Inverse, SelectForHorizontalScrolling, LeftJustify)
        }
    }
}
end     REM **  HighlightCurrentEntry
```

```
REM ************************************************************************
REM **
REM **  Routine Name:   ScrollHistoryUp
REM **
REM **  Description:
REM **      Scroll history up one entry.
REM **
REM ************************************************************************
routine ScrollHistoryUp
{
REM ** If the currently highlighted entry is off the screen, scroll the screen down to
REM ** make room for it.
    if(HighlightedHistoryBufferEntry < HistoryBufferEntryAtScreenTop)
    {
        ScrollDisplay(CharacterHeight)
        HistoryBufferEntryAtScreenTop = HistoryBufferEntryAtScreenTop - 1
        HistoryBufferEntryAtScreenBottom = HistoryBufferEntryAtScreenBottom - 1
    }

end     REM **  ScrollHistoryUp
```

```
REM ************************************************************************
REM **
REM **  Routine Name:   ScrollHistoryDown
REM **
REM **  Description:
REM **      Scroll history down one entry.
REM **
REM ************************************************************************
routine ScrollHistoryDown
{
REM ** If the currently highlighted entry is off the screen, scroll the screen up to
REM ** make room for it.
    if(HighlightedHistoryBufferEntry > HistoryBufferEntryAtScreenBottom)
    {
        ScrollDisplay(-CharacterHeight)
        HistoryBufferEntryAtScreenTop = HistoryBufferEntryAtScreenTop + 1
        HistoryBufferEntryAtScreenBottom = HistoryBufferEntryAtScreenBottom + 1
    }
}
end     REM **  ScrollHistoryDown
```

```
REM ************************************************************************
REM **
REM **  Routine Name:   AddNewDataToHistoryBuffer
REM **
REM **  Description:
REM **      Add the new data (LastDecodedMessage) to the beginning of the
REM **      history buffer and display the most recent entry.  It is assumed
REM **      that the display cursor is at the beginning of the display line
REM **      where data was just entered.
REM **
REM ************************************************************************
routine AddNewDataToHistoryBuffer
{
REM ** If we don't yet have 10 data entries in the history buffer, increment the
REM ** entries in use count.
    if(NumberOfEntriesUsedInHistoryBuffer < 10)
    {
        NumberOfEntriesUsedInHistoryBuffer = NumberOfEntriesUsedInHistoryBuffer + 1
    }

REM ** Shuffle entries 2 to 9 up one level to make room for a new number 2 entry.
REM ** The existing number 10 entry will fall out of the history buffer.
    HistoryBufferEntriesTwoToTen[8] = HistoryBufferEntriesTwoToTen[7]
```

```
    HistoryBufferEntriesTwoToTen[7] = HistoryBufferEntriesTwoToTen[6]
    HistoryBufferEntriesTwoToTen[6] = HistoryBufferEntriesTwoToTen[5]
    HistoryBufferEntriesTwoToTen[5] = HistoryBufferEntriesTwoToTen[4]
    HistoryBufferEntriesTwoToTen[4] = HistoryBufferEntriesTwoToTen[3]
    HistoryBufferEntriesTwoToTen[3] = HistoryBufferEntriesTwoToTen[2]
    HistoryBufferEntriesTwoToTen[2] = HistoryBufferEntriesTwoToTen[1]
    HistoryBufferEntriesTwoToTen[1] = HistoryBufferEntriesTwoToTen[0]


REM ** Copy the most recent history buffer entry into the second history buffer entry.
REM ** Note that if the most recent entry was larger than 64 characters, it will
REM ** automatically be truncated, because each entry in HistoryBufferEntriesTwoToTen
REM ** is 64 characters maximum.  If the length of the most recent entry exceeds 64
REM ** characters, substitute character 64 with a double squiggle (ASCII hex 81) so
REM ** that the user can see that it has been truncated.
    HistoryBufferEntriesTwoToTen[0] = HistoryBufferEntryOne
    if(Length(HistoryBufferEntryOne) > 64)
    {
        HistoryBufferEntriesTwoToTen[0](63,63) = "\81"
    }

REM ** Copy the input data just received into history buffer entry one.
    HistoryBufferEntryOne = LastDecodedMessage
```

```
REM ** Display the most recent entry
    OutputString(HistoryBufferEntryOne)
}
end     REM **  AddNewDataToHistoryBuffer




REM ************************************************************************
REM **
REM **  Routine Name:   ScrollThroughHistoryBuffer
REM **
REM **  Description:
REM **      Allow the user to scroll through the history buffer so long as the
REM **      last key pressed is an up or down arrow.  As soon as any other key
REM **      is pressed, leave it in the keypad buffer and return to the caller.
REM **
REM **      Note that if LastKeyPressed has been set to an arrow key scan code
REM **      (17 or 18), we'll use it to do the first scroll up or down.
REM **
REM ************************************************************************
routine ScrollThroughHistoryBuffer
{
```

```
REM ** Keep looking for an arrow key until the trigger is pressed or the user presses a non-arrow
REM ** key.
    Do
    {
    REM ** No arrow key has been pressed yet.
        ArrowKeyPressed = false

    REM ** Keep looking for a key press until we get one (KeyLook returns non-zero) or the
    REM ** trigger is pressed.  Note that KeyLook will return a key press, but it won't clear
    REM ** it out of the buffer, so if the key pressed isn't an up or down arrow, the key will
    REM ** remain in the keypad buffer for the next InputString.
        While ((NOT (TriggerHit)) AND (LastKeyPressed < 1 ))
        {
            LastKeyPressed = KeyLook
        }


    REM ** Did we get a down arrow?
        if (LastKeyPressed = 17)
        {
        REM ** Flush the arrow key out of the keypad buffer, since KeyLook didn't remove the key
        REM ** from the buffer.
            i = GetKey
```

```
REM ** If there is at least one entry in the history buffer, indicate that an arrow key
REM ** was pressed scroll down.  Otherwise, return to the caller to continue getting
REM ** input.
    if(NumberOfEntriesUsedInHistoryBuffer > 0)
    {
        ArrowKeyPressed = true

    REM ** Scroll down if we aren't at the end of the history buffer.
        if (HighlightedHistoryBufferEntry < NumberOfEntriesUsedInHistoryBuffer)
        {
        REM ** Beep to indicate that we've had a key press.
            Beep(800,16,1)

        REM ** Remove the highlight (if any) from the current entry, move the highlight
        REM ** pointer to the next entry down, scroll the screen, and highlight the next
        REM ** entry.
            Execute(UnhighlightCurrentEntry)
            HighlightedHistoryBufferEntry = HighlightedHistoryBufferEntry + 1
            Execute(ScrollHistoryDown)
            Execute(HighlightCurrentEntry)
        }
    }
}
```

```
REM ** Did we get an up arrow?
    if (LastKeyPressed = 18)
    {
    REM ** Flush the arrow key out of the keypad buffer, since KeyLook didn't remove the key
    REM ** from the buffer.
        i = GetKey

    REM ** Always unhighlight the current entry first.  If the current entry is the first in
    REM ** the history buffer, we're going to stop scrolling and return so that the user can
    REM ** enter new input.
        Execute(UnhighlightCurrentEntry)

    REM ** If the highlighted entry is non-zero, go ahead and do the keyclick, even
    REM ** if we're at the top of the buffer and we're going to return to get more
    REM ** input.  However, don't keyclick if we're already above the top of the buffer.
        if (HighlightedHistoryBufferEntry > 0)
        {
            Beep(800,16,1)
        }

    REM ** Move the highlight pointer to the next entry up.
        HighlightedHistoryBufferEntry = HighlightedHistoryBufferEntry -1

    REM ** If the new highlighted entry is non-zero, indicate that an arrow key was pressed,
```

```
        REM ** scroll the screen, and highlight the next entry.  If the new highlighted entry is
        REM ** zero, indicating that the user just scrolled past the top of the buffer, leave
        REM ** ArrowKeyPressed set to false so that we'll quit the scroll loop and go get more
        REM ** input.
            if (HighlightedHistoryBufferEntry > 0)
            {
                ArrowKeyPressed = true
                Execute(ScrollHistoryUp)
                Execute(HighlightCurrentEntry)
            }
        }

    REM ** Set LastKeyPressed so that if we look for another scroll key, we'll be forced to
    REM ** look for a new key press first.
        LastKeyPressed = 0
    }  While ((NOT (TriggerHit)) AND (ArrowKeyPressed))
}
end     REM **  ScrollThroughHistoryBuffer



REM ************************************************************************
```

```
REM **
REM **  Routine Name:   ResdisplayBeginningEntriesOfHistoryBuffer
REM **
REM **  Description:
REM **      Redisplay the first entries of the history buffer in preparation
REM **      for getting new input.
REM **
REM ************************************************************************
routine ResdisplayBeginningEntriesOfHistoryBuffer
{
REM ** Refresh the display if we're not already at the top of the screen.
    if(HistoryBufferEntryAtScreenTop > 0)
    {
    REM ** If the first history buffer entry currently displayed (HistoryBufferEntryAtScreenTop)
    REM ** will still be visible when we redisplay from the beginning of the buffer, then simply
    REM ** scroll backward from that point to make room for the history buffer entries above it
    REM ** that need to be redisplayed; this will (hopefully) be quicker than rewriting the entire
    REM ** screen.  If the first history buffer entry currently displayed will NOT be visible when
    REM ** we redisplay, then erase the display.
        if(HistoryBufferEntryAtScreenTop < ((DisplayHeight + 1) / CharacterHeight))
        {
            ScrollDisplay(HistoryBufferEntryAtScreenTop * CharacterHeight)
            NumberOfEntriesToRedisplay = HistoryBufferEntryAtScreenTop
        }
```

```
    else
    {
        EraseDisplay
        NumberOfEntriesToRedisplay = ((DisplayHeight + 1) / CharacterHeight)
    }


REM ** Now we'll display the history buffer entries that aren't on the screen, but should be.
    NextEntryToRedisplay = 0
    while(NextEntryToRedisplay < NumberOfEntriesToRedisplay)
    {
    REM ** Before rewriting the next history buffer entry, erase anything on the display that
    REM ** is already there.
        MoveDisplayCursorTo(0, ((NextEntryToRedisplay + 1) * CharacterHeight))
        EraseLine

    REM ** Use the appropriate history buffer depending on whether this is the first entry or
    REM ** entries two to ten.
        if(NextEntryToRedisplay = 0)
        {
            OutputString(HistoryBufferEntryOne, Normal, LeftJustify)
        }
        else
        {
```

```
                OutputString(HistoryBufferEntriesTwoToTen[NextEntryToRedisplay - 1],
                                                    Normal, LeftJustify)
            }

            NextEntryToRedisplay = NextEntryToRedisplay + 1
        }

    REM ** Reset the top and bottom displayed entry pointers.
        HistoryBufferEntryAtScreenTop = 0
        HistoryBufferEntryAtScreenBottom = HistoryBufferEntryAtScreenTop + ((DisplayHeight + 1) /
CharacterHeight) -1
    }

REM ** Unhighlight the current entry if its' still on the display
    Execute(UnhighlightCurrentEntry)
}
end     REM **  ResdisplayBeginningEntriesOfHistoryBuffer




REM ************************************************************************
REM **
REM **  Routine Name:   Main Program
```

```
REM **
REM **  Description:
REM **      Clear the display, then continuously look for a decoded symbol
REM **      and send the result to the host.
REM **
REM ************************************************************************
program
{
REM ** Indicate that there are no entries in the history buffer at the start of
REM ** the program
    NumberOfEntriesUsedInHistoryBuffer = 0

REM ** Start with a clean display
    EraseDisplay

REM ** Continually get input from the keypad or scanner. If data is returned, send
REM ** it on to the host and add it to the history buffer.  If an uparrow or downarrow
REM ** was pressed, allow the user to scroll through the history buffer.
    Forever
    {
    REM ** Set the history buffer pointers to indicate that we're getting input.
        HighlightedHistoryBufferEntry = 0
        HistoryBufferEntryAtScreenTop = 0
        HistoryBufferEntryAtScreenBottom = HistoryBufferEntryAtScreenTop + ((DisplayHeight + 1) /
CharacterHeight) -1
```

```
REM ** Blank out the top line of the display before looking for more input.
    MoveDisplayCursorTo(0, 0)
    EraseLine

REM ** Get either barcode or keypad input, returning immediately if an uparrow (scan
REM ** code 18) or downarrow (scan code 17) is pressed.  If any data is already in the
REM ** keypad buffer, it will be used as part of the input stream.
    LengthOfInput = InputString(LastDecodedMessage, Source=Keypad+Scanner, QuitOnKey=17,
                QuitOnKey=18, PreserveBuffer, DataSource=DataSourceInformation)

REM ** If we received input, let's try to send it to the host.
    if (LengthOfInput > 0)
    {
    REM ** If we successfully send the data to the host, update the history buffer with
    REM ** the new data and display it.
        if(Send(LastDecodedMessage, DataSource=DataSourceInformation))
        {
        REM ** Add new data to the history buffer and scroll the display screen.
            Execute(AddNewDataToHistoryBuffer)

            ScrollDisplay(CharacterHeight())
        }
    }
```

```
    else            rem, gets here if up/down arrow keys
    {
    REM ** If the return code from InputString was -17 (down arrow) or -18
    REM ** (up arrow), scroll the display.
        if((LengthOfInput = -18) OR (LengthOfInput = -17))
        {
            LastKeyPressed = -LastKeyPressed    REM ** Convert the arrow key return code to an
                                                REM **  arrow key scan code, to tell
                                                REM **  ScrollThroughHistoryBuffer that the
                                                REM **  last key press was an up or down arrow.

            Execute (ScrollThroughHistoryBuffer)    REM ** Permit the user to scroll through
                                                    REM **  the history buffer.

            Execute (ResdisplayBeginningEntriesOfHistoryBuffer) REM ** The user has finished
                                                                REM **  scrolling, so we need
                                                                REM **  to refresh the display
                                                                REM **  with the first entries
                                                                REM **  in the history buffer.
        }
    }
}
```

```
}
end     REM **  Main Program
```

Part Number and Quantity Script

The Part Number and Quantity Script prompts for **PART #**. The operator can either scan a bar code representing the part number or type in the part number on the keypad. The part number may be composed of any character. The part number will not be sent to the base until after the quantity is entered. The operator then will be prompted to enter a quantity; only digits are accepted. After entering the quantity, the scanner concatenates the part number, a tab character, and the quantity and transmits the combined message to the base. This ensures that the scanner remains synchronized with the host, and that part numbers and quantities from multiple scanners do not get intermingled.

```
REM ************************************************************************
REM **
REM **  Script Name:    PartNo.vc   (UPC Part # / Qty # sample script for 3875)
REM **
REM **  Description:
REM **      This program will continuously prompt the user for a part number
REM **      quantity.  Each part number / quantity pair entered by the user will
REM **      be formatted into an output message and sent to the host.  The part
REM **      number and quantity may be entered by either keypad or barcode scan.
REM **      The quantity value may be adjusted up or down by pressing the up
REM **      arrow or down arrow respectively.
REM **      While trying to get a quantity value, the user will be allowed to
REM **      revise the part number just entered by pressing the asterisk key.
REM **      The formatted output message will be sent to the host until the send
REM **      is successful or the user decides not to retry a failed message
REM **      send.
REM **
REM **      All prompts are language specific.  The "DSPCTY" menu setting is
```

```
REM **      used to specify the language.  Each type of prompt contains six
REM **      possible translations; selection of the appropriate translation is
REM **      made via the "DSPCTY" setting.
REM **
REM ** (c) Copyright 2002 Hand Held Products, Inc.
REM **
REM ************************************************************************

REM ************************************************************************
REM **
REM **  Revision History
REM **      02/19/02 - Initial release.
REM **
REM **      Source Control Information
REM **          $RCSfile: partno.vc $
REM **          $Revision: 1.9 $
REM **          $Author: HolzhauD $
REM **          $Date: 2002/02/28 18:08:12 $
REM **
REM ************************************************************************



REM ** Variable declarations (note all variables are global in scope)
```

```
STRING  DataSourceInformation*8          REM ** Storage for the 4 byte data source
                                         REM **  returned by InputString.

STRING  CountryCodeReturnString*40       REM ** Holds the string returned when querying
                                         REM **  the value of menu setting DSPCTY.

INT     CountryCode                      REM ** Numeric country code found in the DSPCTY
                                         REM **  menu setting (0 - English, 1 - Spanish,
                                         REM **  2 - Dutch, 3 - French, 4 - German,
                                         REM **  and 5 - Italian).

STRING  PartNumberPrompt*40[6]           REM ** Part number prompts for each possible
                                         REM **  language, where CountryCode is the index
                                         REM **  that determines which PartNumberPrompt
                                         REM **  element to use.

STRING  QuantityPrompt*40[6]             REM ** Quantity value prompts for each possible
                                         REM **  language, where CountryCode is the index
                                         REM **  that determines which QuantityPrompt
                                         REM **  element to use.

STRING  TransmitErrorPrompt*40[6]        REM ** Host transmission error prompts for each
                                         REM **  possible language, where CountryCode is
```

```
                                        REM **  the index that determines which
                                        REM **  TransmitErrorPrompt element to use.

STRING  TransmitRetryPrompt*40[6]       REM ** Host transmission retry prompts for each
                                        REM **  possible language, where CountryCode is
                                        REM **  the index that determines which
                                        REM **  TransmitRetryPrompt element to use.

STRING  YesQueryPrompt*40[6]            REM ** Yes query selection prompts for each
                                        REM **  possible language, where CountryCode is
                                        REM **  the index that determines which
                                        REM **  YesQueryPrompt element to use.

STRING  NoQueryPrompt*40[6]             REM ** No query selection prompts for each
                                        REM **  possible language, where CountryCode is
                                        REM **  the index that determines which
                                        REM **  NoQueryPrompt element to use.

BOOLEAN ReenterPartNumber               REM ** Indicates when user has pressed the up
                                        REM **  key, while entering a quantity value, in
                                        REM **  order to revise the part number previously
                                        REM **  entered.

STRING  PartNumber*2800                 REM ** Part number entered by the user.  2800 bytes
```

```
                                        REM **  are allocated to accommodate the largest
                                        REM **  possible barcode read.

INT     QuantityValue                   REM ** Quantity value entered by the user.

INT     DisplayPixelWidthOfString       REM ** Set to the width in display pixels of a
                                        REM **  string.  Since the text fonts are
                                        REM **  proportional, we need to compute the pixel
                                        REM **  width of each string that will go in a
                                        REM **  prompt box so that we can center it.

STRING  OutputMessage*3000              REM ** Output message to be sent to the host by
                                        REM **  SendOutputMessage routine.

BOOLEAN Retransmit                      REM ** Indicates whether or not to retry transmission
                                        REM **  of a failed output message send operation.

INT     RetryKeyPress                   REM ** Keypad scan code of a key pressed while
                                        REM **  waiting for the user to choose to retry
                                        REM **  or cancel a failed output message
                                        REM **  transmission.

INT     LengthOfQuantityValueInput      REM ** Number of characters in the quantity value
                                        REM **  read by InputNumber
```

```
    INT     i                                   REM ** Index variable
```

```
REM ************************************************************************
REM **
REM **  Routine Name:   GetPartNumber
REM **
REM **  Description:
REM **      Get a part number from the keypad or the display.
REM **
REM ************************************************************************
routine GetPartNumber
{
REM ** Display the language specific part number prompt, scrolling the display to
REM ** make room for the input.
```

```
    Execute(ScrollUp)
    OutputString(PartNumberPrompt[CountryCode])
    Execute(ScrollUp)

REM ** Get the part number.
    InputString(PartNumber, Source=Keypad+Scanner, DataSource=DataSourceInformation)

REM ** Erase the line and display the entered part number so that the user can see it.
    MoveDisplayCursorTo(0, (DisplayHeight + 1 - CharacterHeight))
    EraseLine
    OutputString(PartNumber)
    Beep(3225, 100, 3)
}
end     REM **  GetPartNumber




REM ************************************************************************
REM **
REM **  Routine Name:   GetRevisedPartNumber
REM **
```

```
REM **  Description:
REM **      While looking for a quantity value, the user has indicated that
REM **      they want to revise the part number.  Ask for a new part number,
REM **      using the previously entered part number as the default.
REM **
REM ************************************************************************
routine GetRevisedPartNumber
{
REM ** Display the language specific part number prompt, scrolling the display to
REM ** make room for the input.
    Execute(ScrollUp)
    OutputString(PartNumberPrompt[CountryCode])
    Execute(ScrollUp)

REM ** Get the part number, using the previously entered part number as a default.
    InputString(PartNumber, Source=Keypad+Scanner, DefaultValue = PartNumber,
                            DataSource = DataSourceInformation)

REM ** Erase the line and display the entered part number so that the user can see it.
    MoveDisplayCursorTo(0, (DisplayHeight + 1 - CharacterHeight))
    EraseLine
    OutputString(PartNumber)
    Beep(3225, 100, 3)
}
```

```
end     REM **  GetRevisedPartNumber




REM ************************************************************************
REM **
REM **  Routine Name:   GetQuantity
REM **
REM **  Description:
REM **      Scroll display text up one line to make room for new text at the
REM **      bottom.  If an asterisk is pressed, set the ReenterPartNumber
REM **      flag to indicate that the user wants to revise the part number.
REM **
REM ************************************************************************
routine GetQuantity
{
REM ** Display the language specific quantity prompt, scrolling the display to
REM ** make room for the input.
    Execute(ScrollUp)
    OutputString(QuantityPrompt[CountryCode])
    Execute(ScrollUp)
```

```
REM ** Keep trying to get a quantity value until we get a number or an asterisk is pressed.
    do
    {
    REM ** Erase the input line.
        MoveDisplayCursorTo(0, (DisplayHeight + 1 - CharacterHeight))
        EraseLine

    REM ** Get the quantity value.  Arrow keys will be used for scrolling,
    REM ** we want to quit on the astrisk key (11).  We don't want to quit on a
    REM ** barcode read failure; the only reasons to quit are that a valid number was
    REM ** read or the astrisk key was pressed.
        LengthOfQuantityValueInput = InputNumber(QuantityValue, Source=Keypad+Scanner,
                    UseArrowKeys, QuitOnKey=11, ContinueOnNoDecode,
                    DataSource=DataSourceInformation)
    } while((LengthOfQuantityValueInput <= 0) AND (LengthOfQuantityValueInput <> -11))

    if(LengthOfQuantityValueInput = -11)
    {
    REM ** We quit on an astrisk key, so the user wants to revise the part number.
        ReenterPartNumber = true
    }
    else
    {
```

```
    REM ** We received a valid quantity.  We'll erase the line and display the entered
    REM ** number so that the user can see it.
        ReenterPartNumber = false
        MoveDisplayCursorTo(0, (DisplayHeight + 1 - CharacterHeight))
        EraseLine
        OutputString(String(QuantityValue))
    }
}
end     REM **  GetQuantity




REM ************************************************************************
REM **
REM **  Routine Name:   ScrollUp
REM **
REM **  Description:
REM **      Scroll display text up one line to make room for new text at the
REM **      bottom.
REM **
REM ************************************************************************
```

```
routine ScrollUp
{
REM ** Scroll the display up by one text line and erase the bottom line.
    ScrollDisplay(-CharacterHeight)
    MoveDisplayCursorTo(0, (DisplayHeight + 1 - CharacterHeight))
    EraseLine
}
end     REM **  ScrollUp




REM ************************************************************************
REM **
REM **  Routine Name:   DisplayTransmitErrorMessageAndPrompt
REM **
REM **  Description:
REM **      Display a language specific transmit error message and retry
REM **      prompt.  Boxes with language specific YES and NO labels are
REM **      displayed over the down and up arrow keys and the text must be
REM **      centered in each box.
REM **
REM ************************************************************************
```

```
routine DisplayTransmitErrorMessageAndPrompt
{
REM ** Clear the screen and display the error and retry prompts in the appropriate language.
    EraseDisplay
    MoveDisplayCursorTo (0,0)
    OutputString (TransmitErrorPrompt[CountryCode], VeryClearSmall)
    MoveDisplayCursorTo (0,8)
    OutputString (TransmitRetryPrompt[CountryCode], VeryClearSmall)

REM ** Display the "YES" and "NO" retry prompt boxes.
    LoadImage(20,05,16)
    LoadImage(20,48,16)

REM ** Compute the width of the YES prompt string and move the display cursor to the start
REM ** position of that prompt string so that the prompt string will be centered in the prompt
REM ** box.
    DisplayPixelWidthOfString = 0
    i = 0
    while(i < Length(YesQueryPrompt[CountryCode]))
    {
        DisplayPixelWidthOfString = DisplayPixelWidthOfString +
                            CharacterWidth(VeryClearSmall, YesQueryPrompt[CountryCode](i,i))
        i = i + 1
    }
```

```
    MoveDisplayCursorTo(28 - (DisplayPixelWidthOfString / 2), 20)
    OutputString (YesQueryPrompt[CountryCode], VeryClearSmall)

REM ** Compute the width of the No prompt string and move the display cursor to the start
REM ** position of that prompt string so that the prompt string will be centered in the prompt
REM ** box.
    DisplayPixelWidthOfString = 0
    i = 0
    while(i < Length(NoQueryPrompt[CountryCode]))
    {
        DisplayPixelWidthOfString = DisplayPixelWidthOfString +
                            CharacterWidth(VeryClearSmall, NoQueryPrompt[CountryCode](i,i))
        i = i + 1
    }
    MoveDisplayCursorTo(70 - (DisplayPixelWidthOfString / 2), 20)
    OutputString (NoQueryPrompt[CountryCode], VeryClearSmall)
}
end     REM **  Routine Name:   DisplayTransmitErrorMessageAndPrompt
```

```
REM ************************************************************************
REM **
REM **  Routine Name:   SendOutputMessage
REM **
REM **  Description:
REM **      Try to send the output message to the host.  If the send fails,
REM **      inform the user and ask the user to choose whether to retry or
REM **      to continue on.  Note that a language translation specific "YES"
REM **      and "NO" prompt for retransmission will be displayed directly
REM **      above the down and up arrow keys, where a down arrow key press
REM **      stipulates a retry and an up arrow key press stipulates that we
REM **      continue on.
REM **
REM ************************************************************************
routine SendOutputMessage
{
    Retransmit = false          REM ** No transmit errors yet, so we'll assume we don't need
                                REM **  to retransmit.

REM ** Try to send the message
    if(NOT (Send(OutputMessage, DataSource=DataSourceInformation)))
    {
    REM ** The send was not successful.  Prompt the user to retry.
        Execute(DisplayTransmitErrorMessageAndPrompt)
```

```
REM ** Keep looking for a key press until the user presses the
REM ** up or down arrow key.
    do
    {
        RetryKeyPress = GetKey
    } while((RetryKeyPress <> 17) AND (RetryKeyPress <> 18))


REM ** If the user pressed down arrow (YES), tell the calling routine to
REM ** retransmit.
    if(RetryKeyPress = 17)
    {
        Retransmit = true
    }

REM ** If the user pressed up arrow (NO), tell the calling routine to
REM ** not to retransmit.
    if(RetryKeyPress = 18)
    {
        Retransmit = false
    }

REM ** Erase the display before looking for another part number
```

```
        EraseDisplay
    }
}
end     REM **  SendOutputMessage




REM ************************************************************************
REM **
REM **  Routine Name:   GetCountryCode
REM **
REM **  Description:
REM **      Get the country code value from the menu setting DSPCTY.
REM **
REM ************************************************************************
routine GetCountryCode
{
REM ** Read the DSPCTY menu setting and extract the 7th character as the country code.
    CountryCodeReturnString = MenuCommand("DSPCTY?!")
    CountryCode = Int(CountryCodeReturnString(6,6))

REM ** If CountryCode is not in the range of 0 to 5, set it to 0 (English) as the default.
```

```
    if((CountryCode < 0) OR (CountryCode > 5))
    {
        CountryCode = 0
    }
}
end     REM **  GetCountryCode




REM ************************************************************************
REM **
REM **  Routine Name:   InitializePrompts
REM **
REM **  Description:
REM **      Initialize the PartNumberPrompt, QuantityPrompt, TransmitErrorPrompt,
REM **      TransmitRetryPrompt, YesQueryPrompt, and NoQueryPrompt string
REM **      arrays with the prompts for each possible country code value.
REM **
REM ************************************************************************
routine InitializePrompts
{
```

```
PartNumberPrompt[0] = "Part#:"
PartNumberPrompt[1] = "N. Parte"
PartNumberPrompt[2] = "Artikel:"
PartNumberPrompt[3] = "Article:"
PartNumberPrompt[4] = "Produkt:"
PartNumberPrompt[5] = "Articolo"

QuantityPrompt[0] = "Qty:"
QuantityPrompt[1] = "Cant.:"
QuantityPrompt[2] = "Aantal"
QuantityPrompt[3] = "Nombre:"
QuantityPrompt[4] = "Anzahl:"
QuantityPrompt[5] = "Quaniti\e0"

TransmitErrorPrompt[0] = "Transmit Error"
TransmitErrorPrompt[1] = "Error"
TransmitErrorPrompt[2] = "geen verbinding"
TransmitErrorPrompt[3] = "Erreur. Voulez"
TransmitErrorPrompt[4] = "keine Verbindung"
TransmitErrorPrompt[5] = "Errore trasm.ne."

TransmitRetryPrompt[0] = "Want to Retry?"
TransmitRetryPrompt[1] = "\bfEnvia otra v\e9z?"
TransmitRetryPrompt[2] = "Herhalen?"
```

```
    TransmitRetryPrompt[3] = "vous r\e9essayer?"
    TransmitRetryPrompt[4] = "Nochmals?"
    TransmitRetryPrompt[5] = "Riprovare?"

    YesQueryPrompt[0] = "Yes"
    YesQueryPrompt[1] = "Si"
    YesQueryPrompt[2] = "Ja"
    YesQueryPrompt[3] = "Oui"
    YesQueryPrompt[4] = "Ja"
    YesQueryPrompt[5] = "Si"

    NoQueryPrompt[0] = "No"
    NoQueryPrompt[1] = "No"
    NoQueryPrompt[2] = "Nee"
    NoQueryPrompt[3] = "Non"
    NoQueryPrompt[4] = "Nein"
    NoQueryPrompt[5] = "No"
}
end     REM **  InitializePrompts
```

```
REM ************************************************************************
REM **
REM **  Routine Name:   Main Program
REM **
REM **  Description:
REM **      Get the country code, initialize the various prompts for each
REM **      language, and then continually look for a part number / quantity
REM **      combination to send to the host.
REM **
REM ************************************************************************
program
{
    Execute(GetCountryCode)         REM ** Get the country code so we'll know what language to use
                                    REM **  for prompts.

    Execute(InitializePrompts)      REM ** Setup the part number, quantity, and host transmission
                                    REM **  prompts for each language.

REM ** Prompt the user for a part number and try to get one.
    EraseDisplay
    Execute(GetPartNumber)
```

```
REM ** Continually look for a part number / quantity combination to send to the host.
    Forever
    {
        DisplayCursorOff

    REM ** Prompt the user for a quantity value and try to get one.
        Execute(GetQuantity)

        if(ReenterPartNumber)
        {
        REM ** User backed out of quantity input; allow the user to revise the part number.
            EraseDisplay
            Execute(GetRevisedPartNumber)
        }
        else
        {
        REM ** User entered valid quantity.  Put together the output message consisting of the
        REM ** part number, TAB character, quantity.
            OutputMessage = PartNumber + "\09" + String(QuantityValue)

        REM ** Keep trying to transmit until we succeed.
            do
            {
                Execute(SendOutputMessage)
```

```
            } while(Retransmit)

        REM ** Prompt the user for another part number and try to get one.
            Execute(GetPartNumber)
        }

    }
}
end     REM **  Main Program
```

7 Glossary

AIM Identifier – A 3 digit code, optionally reported between the global preamble and edited data, that identifies the symbology of the decoded symbol and may help to identify characteristics unique to that symbology. For more information, see AIM guidelines.

Conditional – For an IF conditional, the part of the IF that determines which course of action to take. For a WHILE or DO WHILE loop, the part that determines whether to run through the loop again.

Default Postamble – a VersaCode command used to specify default fixed text conditionally reported as the trailer for a Send command, provided that a postamble is not specified in the Send that overrides the default postamble.

Default Preamble – a VersaCode command used to specify default fixed text conditionally reported as the leader for a send command, provided that a preamble is not specified in the Send that overrides the default preamble.

Flagged Error – When this command is encountered in a VersaCode program, the scanner performs some platform-specific error handling. Generally, the scanner handles this command the same way it handles a read error.

Flagged Termination – When this command is encountered in a VersaCode program, the scanner performs some platform-specific error handling. Generally, the scanner handles this by terminating the VersaCode program.

Flagged Warning – When this command is encountered in a VersaCode program, the scanner performs some platform-specific warning handling. Generally, the scanner handles this command the same way it handles a read warning condition.

Menu Settings – Configuration settings that affect scanner operation, communication, and decoding.

Postamble – Fixed text that is conditionally reported as the trailer to a Send command.

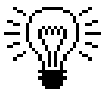
Preamble – Fixed text that is conditionally reported as the leader to a Send command.

Host Commands - Host VersaCode commands are a subset of VersaCode commands that get input, write to the display, and do other I/O or system related activities. They do not support variables, but otherwise are syntactically equivalent to their VersaCode programming language counterparts.

Symbol Data – ASCII data result of reading a symbol, before editing.

Bitmap Chart

The following bitmaps are available through the use of LoadImage (see page 4-45):

			Size in Pixels	
No.	Image Name	Graphic	Width	Height
1	Antenna		10	18
2	ArrowDown		7	5
3	ArrowLeft		6	8
4	ArrowRight		6	8
5	ArrowUp		7	5
6	AssocInProg		58	26
7	BackLightOff		31	26
8	BackLightOn		31	26
9	Barlarge		3	16
10	BarMedium		3	16
11	BarSmall		3	16
12	BarXLarge		3	16
13	BarXSmall		3	16

			Size in Pixels	
No.	Image Name	Graphic	Width	Height
14	BatteryHigh		11	18
15	BatteryLow		11	18
16	Border		97	32
17	BulletRound		5	5
18	BulletSquare		5	5
19	Busy		15	24
20	ButtonBlank		44	16
21	ButtonInverse		44	16
22	ButtonCancel		47	16
23	ButtonNo		32	16
24	ButtonOK		47	16
25	ButtonYes		32	16
26	Clock		21	21
27	Contrast		26	26

			Size in Pixels	
No.	Image Name	Graphic	Width	Height
28	FaceSad		26	26
29	FaceSmiley		26	26
30	Failure		26	26
31	FunctionKey	FN	18	9
32	LogoHHP		97	32
33	LogoIT3875		96	32
34	Success		26	26
35	ThumbDown		21	26
36	ThumbUp		21	26
37	TruncationSign	⁑	6	8
38	Warning		32	26

IT3875 Font Characters Chart

Dec.	Hex	Char	Dec.	Hex	Char	Dec.	Hex	Char	Dec.	Hex	Char
0	00	NUL	32	20		64	40	@	96	60	‘
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	“	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	‘	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	

Dec.	Hex	Char	Dec.	Hex	Char	Dec.	Hex	Char	Dec.	Hex	Char
128	80	€	160	A0		192	C0	À	224	E0	à
129	81		161	A1	¡	193	C1	Á	225	E1	á
130	82	‚	162	A2	¢	194	C2	Â	226	E2	â
131	83	*ƒ*	163	A3	£	195	C3	Ã	227	E3	ã
132	84	„	164	A4	¤	196	C4	Ä	228	E4	ä
133	85	…	165	A5	¥	197	C5	Å	229	E5	å
134	86	†	166	A6	¦	198	C6	Æ	230	E6	æ
135	87	‡	167	A7	§	199	C7	Ç	231	E7	ç
136	88	ˆ	168	A8	¨	200	C8	È	232	E8	è
137	89	‰	169	A9	©	201	C9	É	233	E9	é
138	8A	Š	170	AA	ª	202	CA	Ê	234	EA	ê
139	8B	‹	171	AB	«	203	CB	Ë	235	EB	ë
140	8C	Œ	172	AC	¬	204	CC	Ì	236	EC	ì
141	8D	□	173	AD	-	205	CD	Í	237	ED	í
142	8E	Ž	174	AE	®	206	CE	Î	238	EE	î
143	8F	□	175	AF	¯	207	CF	Ï	239	EF	ï
144	90	□	176	B0	°	208	D0	Ð	240	F0	ð
145	91	‘	177	B1	±	209	D1	Ñ	241	F1	ñ
146	92	’	178	B2	²	210	D2	Ò	242	F2	ò
147	93	“	179	B3	³	211	D3	Ó	243	F3	ó
148	94	”	180	B4	´	212	D4	Ô	244	F4	ô
149	95	•	181	B5	µ	213	D5	Õ	245	F5	õ
150	96	–	182	B6	¶	214	D6	Ö	246	F6	ö
151	97	—	183	B7	·	215	D7	×	247	F7	÷
152	98	˜	184	B8	¸	216	D8	Ø	248	F8	ø
153	99	™	185	B9	¹	217	D9	Ù	249	F9	ù
154	9A	š	186	BA	º	218	DA	Ú	250	FA	ú
155	9B	›	187	BB	»	219	DB	Û	251	FB	û
156	9C	œ	188	BC	¼	220	DC	Ü	252	FC	ü
157	9D	□	189	BD	½	221	DD	Ý	253	FD	ý
158	9E	ž	190	BE	¾	222	DE	Þ	254	FE	þ
159	9F	Ÿ	191	BF	¿	223	DF	ß	255	FF	ÿ

Numeric Key Codes

The following table lists the IT3875 keypad and the numeric key codes assigned to each key and each FN-key sequence. These are the numeric key codes, not the character that appears on the key itself. In the case of a FN-key sequence, the FN key is pressed and released and then another key is pressed.

Key	Returns	Function Key	Returns
1	1	1	65
2	2	2	66
3	3	3	67
4	4	4	68
5	5	5	69
6	6	6	70
7	7	7	71
8	8	8	72
9	9	9	73
0	10	0	74
*	11	down arrow	81
Enter	16	up arrow	82
down arrow	17		
up arrow	18		
backspace	19		

Quick Reference Chart

Function	Return Value	Host Command	Page
Beeper and LEDs			
Beep([Frequency =]user-defined numeric expression, [Duration =]user-defined numeric expression, [Volume =]user-defined numeric expression)	None	BEP	4-2
IndicatorBad	None	IBS	4-28
IndicatorGood	None	IGS	4-29
IndicatorLightOff	None	ILD	4-29
IndicatorLightOn([Red | Green | Yellow], [[OnCycleTime=]user-defined numeric expression 1], [[NumberOfCycles=]user-defined numeric expression 2],)	None	ILE	4-30
Data Format			
CompareNextCharacterTo('x')	bool bMatch	N/A	5-1
IsNextCharacterNonNumeric	bool bCompare	N/A	5-1

Function	Return Value	Host Command	Page
IsNextCharacterNumeric	bool bCompare	N/A	5-2
MoveCursorAheadBy(n)	None	N/A	5-2
MoveCursorAheadToChar(‘x’)	None	N/A	5-2
MoveCursorAheadToNonChar(‘x’)	None	N/A	5-3
MoveCursorBackBy(n)	None	N/A	5-3
MoveCursorBackToChar(‘x’)	None	N/A	5-4
MoveCursorBackToNonChar(‘x’)	None	N/A	5-4
MoveCursorToBeginning	None	N/A	5-4
MoveCursorToEnd	None	N/A	5-5
ReplaceCharacters(‘x1a’ with ‘x1b’, ‘x2a’ with ‘x2b’, … ‘xna’ with ‘xnb’)	None	N/A	5-5
SuppressCharacters(‘x1’, ‘x2’, … ‘xn’)	None	N/A	5-5
TurnOffCharacterReplacement	None	N/A	5-6
TurnOffCharacterSuppression	None	N/A	5-6
WriteAllButLastCharactersStartingAtCursor(n)	None	N/A	5-6

Function	Return Value	Host Command	Page
WriteCharactersFromCursor (n, [[End=]'x'])	None	N/A	5-7
WriteCharactersFromCursorToChar('x', [[End=]'y'])	None	N/A	5-7
WriteCharactersFromCursorToEnd([[End=]'x'])	None	N/A	5-8
WriteCharMultipleTimes('x', n times)	None	N/A	5-8
Display			
CharacterHeight([VeryClearSmall | VeryClearMedium | VeryClearLarge])	int iPixels	CHH	4-8
CharacterWidth([VeryClearSmall | VeryClearMedium | VeryClearLarge], user-defined string expression)	int iPixels	CHW	4-10
DisplayBacklightOff	None	BKD	4-13
DisplayBacklightOn	None	BKE	4-14
DisplayCursorOff	None	CURD	4-15
DisplayCursorOnBlink	None	CURB	4-15
DisplayCursorOnSteady	None	CURS	4-16
DisplayHeight	int iPixels	DSPH	4-16

Function	Return Value	Host Command	Page
DisplayLanguage	"English," "Spanish," "Dutch," "French," "German," "Italian," or "Unknown"	DSPLNG	4-17
DisplayMenu ([MenuStart=(user-defined numeric expression x, user-defined numeric expression y),] [Title=user-defined string expression 1[([No]Scroll)],] [MenuItem=string expression 2 [([Hotkey=user-defined string expression 3,] [DefaultSelection])],] [WrapOnSelectionScrolling,] [MaximumDisplayLinesToUse=user-defined numeric expression 3,] [TimeoutOnNoSelection=user-defined numeric expression 4,] [QuitOnBackspace])	int iListItem Number or string sListItem	DMN	4-17
DisplayWidth	int iPixels	DSPW	4-20
EraseDisplay	None	EDSP	4-20
EraseLine([[x=]user-defined numeric expression 1, [y=]user-defined numeric expression 2])	None	ELN	4-21
GetFont	int iFontNum	GFNT	4-24
LoadImage([Image=]user-defined numeric expression 1 | Antenna | ArrowDown | ArrowLeft | ArrowRight | ArrowUp | AssocInProg | BackLightOff | BackLightOn | BarLarge | BarMedium | BarSmall | BarXLarge | BarXSmall | BatteryHigh | BatteryLow | Border | BulletRound | BulletSquare | Busy | ButtonBlank | ButtonInverse | ButtonCancel | ButtonNo | ButtonOK | ButtonYes | Clock | Contrast | FaceSad | FaceSmiley | Failure | FunctionKey | LogoHHP | LogoIT3875 | Success | ThumbDown | ThumbUp | TruncationSign | Warning, [x=]user-defined numeric expression 2, [y=]user-defined numeric expression 3)	bool bSuccess	LDG	4-45

Function	Return Value	Host Command	Page
MoveDisplayCursorTo([x=]user-defined numeric expression 2, [y=]user-defined numeric expression 2)	bool bSuccess	CUR	4-48
OutputString([String=]user-defined string expression, [[Video=]Normal | Inverse], [[Font=]user-defined numeric expression | VeryClearSmall | VeryClearMedium | VeryClearLarge], [SelectForHorizontalScrolling], [LeftJustify | RightJustify])	bool bSuccess	OUTS	4-49
ScrollDisplay (user-defined numeric expression)	bool bSuccess	SCRL	4-54
SetFont(VeryClearSmall | VeryClearMedium | VeryClearLarge)	None	SFNT	4-57
TemporarilyEnableBacklight([Time=]user-defined numeric expression)	None	TBLE	4-61
Input and Keypad			
GetKey	Scan code of key code read. See page 4-26 for list.	GETKY	4-25
InputNumber([Number=]user-defined numeric variable, [[Source=] Keypad | Scanner | Host | All], [[Default-Value=]user-defined numeric expression 1], [UseArrowKeys | IgnoreArrowKeys], [QuitOnKey=user-defined numeric expression 2], [ContinueOnNoDecode | QuitOnNoDecode], [CursorOn | CursorOff], [SilentInput], [RefreshDisplayAfterInput], [FlushBuffer | PreserveBuffer], [[DataSource=]user-defined string variable], [[Mini-mum=]user-defined numeric expression 3], [[Maximum=]user-defined numeric expression 4], [[Timeout=]user-defined numeric expression 5])	bool bSuccess	INN	4-31

Function	Return Value	Host Command	Page
InputString([String=]user-defined string pointer, [[Source=] Keypad | Scanner | Host | All], [[DefaultValue=]user-defined string expression], [QuitOnKey=user-defined numeric expression 2], [ContinueOnNoDecode | QuitOnNoDecode], [CursorOn | CursorOff], [SilentInput], [RefreshDisplayAfterInput], [FlushBuffer | PreserveBuffer], [[DataSource=]user-defined string variable], [[Minimum=]user-defined numeric expression 3], [[Maximum=]user-defined numeric expression 4], [[Timeout=]user-defined numeric expression])	int iStringLength	INS	4-36
KeyLook	Scan code of key code read. See page 4-42 for list.	KLK	4-41
WaitForKey ([buffer control,] [timeout=user-defined numeric expression])	Scan code of key code read. See page 4-66 for list.	WAITKY	4-65
Menu			
MenuCommand(user-defined string expression)	bool bSuccess	MNU	4-47
Receive Host Command			
CheckForCommand([[HostRoutines=]NONE | HOST | (list)])	None	N/A	4-11
WaitForCommand([[HostRoutines=]NONE | HOST | (list)])	None	N/A	4-64
WaitForMultipleCommands([[HostRoutines=]NONE | HOST | (list)])	None	N/A	4-67

Function	Return Value	Host Command	Page
Send			
DefaultPrefix (user-defined string expression, [Always | IfData | IfNoData])	None	N/A	4-12
DefaultSuffix (user-defined string expression, [Always | IfData | IfNoData])	None	N/A	4-12
Send([String=]user-defined string expression 1, [[Prefix=]DefaultPrefix | PrefixAlways(user-defined string expression)], [[Suffix=]DefaultSuffix | SuffixAlways(user-defined string expression))], [[DataSource=]user-defined string expression 2], [UseIndicatorFeedback | DoNotUseIndicatorFeedback])	bool bSuccess	N/A	4-55
String			
CaseInsensitiveCompare(user-defined string 1 operator user-defined string 2)	bool bComparison	N/A	4-3
CaseInsensitiveWildcardCompare(user-defined string 1, user-defined string 2)	bool bMatch	N/A	4-4
CaseSensitiveCompare (user-defined string 1 operator user-defined string 2)	bool bComparison	N/A	4-6
CaseSensitiveWildcardCompare(user-defined string 1, user-defined string 2)	bool bMatch	N/A	4-7
CharToInt (char cChar)	int iASCIIDecimalVal	N/A	2-15
IntToChar (int iASCIIDecimalVal)	char cChar	N/A	2-15
LeftJustify([String=]user-defined string, [Total=] n, [[Fill=]'x'])	string sResult	N/A	4-43
Length(user-defined string expression)	int iLength	N/A	4-44

Function	Return Value	Host Command	Page
LowerCase(user-defined string expression)	string sResult	N/A	4-47
RightJustify([String=]user-defined string, [Total=]n, [[Fill=]'x'])	string sResult	N/A	4-52
SetLength (user-defined string variable, user-defined numeric expression)	bool bSuccess	N/A	4-58
StringComparisonsAreCaseInsensitiveByDefault	None	N/A	4-59
StringComparisonsAreCaseSensitiveByDefault	None	N/A	4-60
UpperCase(user-defined string expression)	string sResult	N/A	4-62
WildcardCompare(user-defined string expression 1, user-defined string expression 2)	bool bMatch	N/A	4-68
Timing			
GetTimer0	int iTimer	N/A	4-27
ResetTimer0	None	N/A	4-51
Wait(user-defined numeric expression)	None	N/A	4-63
Trigger			
TriggerHit	bool bTrigger	TRG	4-62
System			

Function	Return Value	Host Command	Page
BatteryStrength	0 to 100	BATLEV	4-2
InterpreterVersion	string	VERIN	4-40
RadioStrength	integer	RADSIG	4-51
ScannerID	string	SCNID	4-53
Workgroup	integer	GRP	4-70

Upgrade your image.

700 Visions Drive
P.O. Box 208
Skaneateles Falls, NY 13153-0208