ITScriptNet®
INDAGO

Version 4

Create Powerful Data Collection
Solutions - Simply and Easily

ITScriptNet Indago Developer Guide

Z-SPACE
TECHNOLOGIES, INC.

ITScriptNet Indago Developer Guide

Table of Contents

Part I

1 Introduction

Welcome to ITScriptNet® Indago, the easy-to-use software that allows you to quickly and easily create data collection solutions for mobile computers. ITScriptNet Indago is designed to be easy-to-use, yet powerful enough to support the most sophisticated applications.

ITScriptNet supports a variety of mobile computers. For a complete listing of supported devices, please refer to our web site http://www.z-space.com.

1.1 Software License Agreement

READ THIS BEFORE USING THE NOTED PROGRAMS
Thank you for selecting ITScriptNet® from BCA Innovations, LLC ("BCAI"). Please read the following License Agreement below before registering the serial number. If you do not accept these terms, return the product unregistered with proof of purchase to the point of purchase for a complete refund. Only if you accept these terms should you register the software.

The enclosed copy of ITScriptNet® is never sold. It is licensed by BCAI to the original customer and to any subsequent licensee of his or her for use in accordance with the terms set forth below. BY REGISTERING THIS SOFTWARE YOU ARE INDICATING ACCEPTANCE OF THESE TERMS. Otherwise, you may return the software and User Guide within ten (10) days to the place where you obtained it for a full refund. Under the terms of this license agreement:

YOU MAY:
For PC-Based Licenses:
1. *Load and use* the software on any computer as long as it is used on only one computer by one user at a time. The software serial number can only be registered once on a single computer. The license cannot be shared over a network. If more than one computer requires the use of the software, then additional license fees will be required for each computer.
2. *Communicate* with the Remote Host Server (ITScriptNet® OMNI™ Server) residing on the host computer with only the number of terminals as there are terminal licenses registered on the host computer. Additional terminal licenses can be purchased and added to the host computer to increase the number of terminals that can be configured to communicate with the host computer. Communication by a terminal with the host computer can be carried by a network and does not violate item 1) above. [This Provision applies to the ITScriptNet® OMNI™ edition only.]
3. *Move* a registered license from one computer to another by unregistering the license from the licensed computer via the method provided in the software, then re-registering the license on a different computer. Compliance with paragraph 1 (Load and Use) above must be maintained. There are no restrictions as to the number of times a license can be registered and unregistered.

For Device-Based Licenses:
4. *Load and Use* the Runtime communication software on any number of computers without needing a PC-based license.
5. *License* each device that will be communicating with a computer. Each device license can be registered on a single terminal. Once registered on a terminal, a Device License cannot be removed or assigned to a different terminal. A terminal with a Device License can communicate with any computer running the Runtime communication software.

For All Licenses:
6. *Copy* the software for back-up purposes only. You may make up to three (3) copies of the software for backup
purposes. All copies must contain the copyright notice printed on the label of the media containing the original copy of the software.
7. *Transfer* the software and license permanently to another person if that person agrees to accept all of the terms and conditions of this Agreement. If you transfer the software, you must at the same time either transfer all copies of the software to the same person or destroy any copies not transferred.
8. *Terminate* this license by destroying the original and all copies of the software in whatever form.

YOU MAY NOT:
1. Loan, rent, lease, give, sublicense or otherwise transfer the software (or any copy), in whole or in part, to any other person, except as noted in paragraph 7 (Transfer) above.
2. Copy or translate the User Guide included with the software.
3. Copy, alter, translate, decompile, or reverse engineer the software, including but not limited to, modify the software to make it operate on non-compatible hardware.
4. Remove, alter or cause not to be displayed, any copyright notices or startup messages contained in the

programs or documentation.

THIS LICENSE WILL TERMINATE AUTOMATICALLY if you fail to comply with the terms and conditions set forth above.

Term
The license is effective until terminated. You may terminate it at any time by destroying the programs together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the programs together with all copies, modifications and merged portions in any form.

Limited Warranty

What is covered?
BCAI warrants to the original customer that (i) the files that comprise the delivery of the software are free from defects in materials and workmanship under normal use, and (ii) the software will perform substantially in accordance with the provided User Guide, if any, or the custom proposal. EXCEPT AS SPECIFIED IN THIS PARAGRAPH, THERE ARE NO
WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND THE PROGRAMS, DOCUMENTATION AND OTHER FILES ARE PROVIDED “AS IS.”
(Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.)

How long does this warranty last?
This Limited Warranty continues for sixty (60) days from the date of delivery of the software to the original customer (“Warranty Period”).

What will BCAI do?
1. BCAI will replace any files which prove defective in materials or workmanship, if you notify BCAI during the Warranty Period with a dated proof of purchase.
2. BCAI will, at its option, either replace the files or correct any software that does not perform substantially in accordance with the provided User Guide or custom proposal if, during the Warranty Period, (i) you notify BCAI in writing of any claimed defects in the software, (ii) and BCAI is able to duplicate the defects on its computer system.
3. If BCAI is unable to replace a defective file or if BCAI is unable to provide corrected software within a reasonable time, BCAI will, at its option, either replace the software with functionally equivalent software or refund the license fees paid by you. THESE ARE YOUR SOLE AND EXCLUSIVE REMEDIES for any and all claims that you may have against BCAI arising out of or in connection with this product, whether made or suffered by you or another person and whether based in contract or tort.
4. IN NO EVENT WILL BCAI BE LIABLE TO YOU OR ANY OTHER PARTY FOR DIRECT, INDIRECT, GENERAL, SPECIAL, INCIDENTAL, CONSEQUENTIAL, EXEMPLARY OR OTHER DAMAGES ARISING FROM THE USE OF OR INABILITY TO USE THE SOFTWARE OR FROM ANY BREACH OFTHIS WARRANTY, EVEN IF BCAI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. (Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above exclusion or limitation may not apply to you.) In no event shall BCAI’s total liability exceed the amount you paid in license fees for the right to use a single copy of this software. BCAI’s software pricing reflects the allocation of risk and limitations on liability contained in this Limited Warranty.

What additional provisions should I be aware of?
1. Because it is impossible for BCAI to know the purposes for which you acquired this software or the uses to which you will put this software, you assume full responsibility for the selection of the software, and for its installation and use and the results of that use.
2. While every reasonable effort has been made to insure that you will receive software that you can use and enjoy, BCAI does not warrant that the functions of the software will meet your requirements or that the

operation of the software will be uninterrupted or error free. Due to the complex nature of computer programs, the programs in this package (like all large programs) will probably never be completely error free.
3. This Limited Warranty does not cover any file which has been the subject of abuse or damages, nor does it cover any software which has been altered or changed by anyone other than BCAI.
4. BCAI is not responsible for problems caused by changes in the operating characteristics of the hardware or operating system software you are using which are made after the release date of this version of ITScriptNet ® with any other software.
5. You agree to comply with all applicable international and national laws that apply to the products as well as end-user, end-use and destination restrictions issued by governments.
6. If the SOFTWARE is labeled as an upgrade, you must be properly licensed to use a product identified by BCAI
as being eligible for the upgrade in order to use the software. Software labeled as an upgrade replaces and will disable the original software which was initially loaded on the computer. After upgrading, you may no longer use the software that formed the basis for your upgrade eligibility. You may use the resulting upgraded product only in accordance with the terms of this license agreement and only with a computer that has also registered the original software.
7. This agreement constitutes the entire agreement between you and BCAI and supersedes any prior understandings and agreements, either oral or written. It shall be interpreted under the laws of the State of Florida.
8. This warranty gives you specific rights and you may also have other rights which vary from state to state.
9. No action for breach of warranty may be commenced more than one (1) year following the expiration date of the above Limited Warranty.
Should you have any questions concerning this Agreement, you may contact BCAI by writing to BCA Innovations, LLC, 8813 NW 23rd Street, Miami, FL 33172.

1.2 Technical Support

If you need technical support on this product, please contact your reseller or hardware manufacturer.

You can also access technical support at http://www.z-space.com, or by emailing support@z-space.com, or by calling (440) 899-7370 between the hours of 9:00 a.m. to 5:00 p.m. EST.

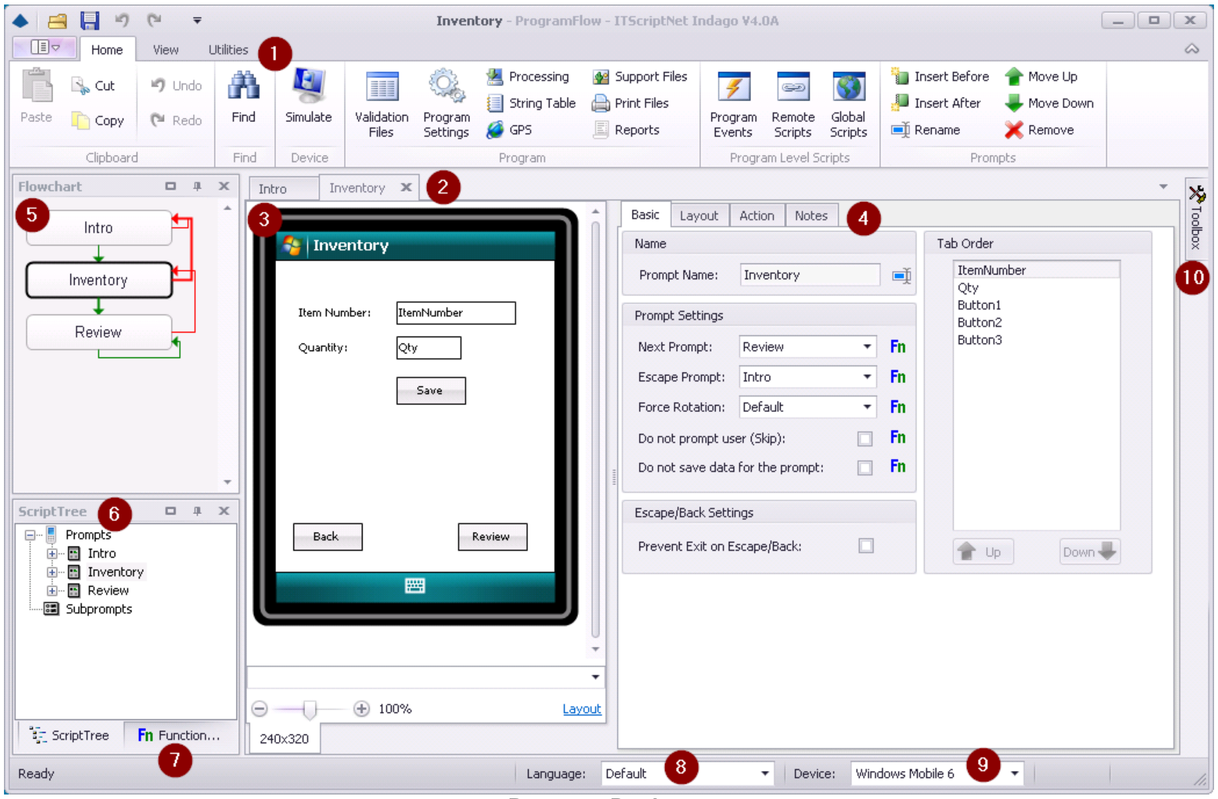
ITScriptNet Web Site
http://www.z-space.com

ITScriptNet Knowledge Base
http://www.z-space.com/kb

Part II

2 Program Designer Tour

This topic describes the major Program Designer program areas.



Name

Prompt Name: Inventory

Prompt Settings

Next Prompt: Review

Escape Prompt: Intro

Force Rotation: Default

Do not prompt user (Skip): ☐

Do not save data for the prompt: ☐

Escape/Back Settings

Prevent Exit on Escape/Back: ☐

Tab Order

ItemNumber
Qty
Button1
Button2
Button3

Up Down

Language: Default

Device: Windows Mobile 6

Program Designer

1. The Ribbon. This is the main menu for the program designer, and contains buttons for all of the major operations that you can do while designing a program. The available options will change to reflect the currently selected prompt or element.
2. The Program Designer allows you to have multiple windows open simultaneously. Each window has a Tab so you can quickly and easily switch between them
3. This is the main Prompt Layout area. You drag and resize elements on the prompt surface to build your layout.
4. This is the properties are for the currently select prompt or element. The tabs group the settings into groups of similar functionality.
5. The Prompt flowchart. This represents the default program flow.
6. The Script Tree. This tree lists all prompts, subprompts, and elements. You may find it easier to navigate through your program design using this tree instead of the flowchart.
7. The Function Reference. This window lists all in-prompt script functions and descriptions for easy reference.
8. Language Selector. If you are designing your program in multiple languages, this selector allow you to choose the language.
9. Device Type selector. This changes the device type view for the prompt layout.
10. The element Toolbox. This pop-out window holds the possible elements that can be added to a program.

Part III

3 Program Design

Prompts
The data collection programs that ITScriptNet Indago creates are based on Prompts. Each program contains a series of one or more Prompts. The program flows from one Prompt to the other as the user enters data. Every Prompt will contain one or more Elements that collect data or are used for display.

Looping
Generally, Prompts for data collection are presented to the user one after another in a logical order. The order of the Prompts within the program determines the order of the Prompts presented to the user at data collection time. When the user gets to the end of the Prompts, the data collected is saved and the user will need to start again at the first Prompt to collect another set of data. Each pass through the Prompts generates a record in the stored data. When downloaded to a text file, each record will be represented by one row of text. When downloaded to a text file or database, each record of collected data corresponds to a line in the file or a record in a database.

Saving Collected Data
A record is saved to Collected Data when the last prompt in the program is Accepted. Alternately, call SaveCollectedData in an in-prompt script to save a collected data record.

In-Prompt Scripts
Most properties of prompt and elements have an associated Function button. You can write a script that overrides the property at runtime. There are also Events for prompts and elements that allow you to execute script code at a particular time.

3.1 Writing Scripts

In-Prompt Script Components

The syntax of scripts in ITScriptNet Indago is easy to learn. Every script is composed of Literal Strings, Constants, Functions, Variables, and Operators.

Using Literal Strings in Scripts

A Literal String is a fixed piece of data. Literal strings are always enclosed in double-quotes. For example, “Description”, “QtyPrompt” and “Data Invalid” are all Literal Strings. You can use Literal Strings in expressions, assign them to variables, or assign them to responses.

Using Constants in Scripts

ITScriptNet includes many Constants which can be used in scripts. Constants are names that are translated into a value. Constants have been defined to translate barcode symbologies, input sources, True and False, etc. For example, you can use TRUE instead of 1, and FALSE instead of 0. These constants may be used as the parameters of a function or comparison. Using constants makes the script more readable and does not require you to know the underlying values. The Constants are located in the tree for easy access on the **Edit Script** screen where they are grouped according to their category. Many of the Constants are related to a function or a set of functions. For example, the "ResponseSource( )" function will return a value indicating whether the user entered the response via keyboard, scanner, image, etc. If you need to test the response source to see if the response was entered on the keypad, you could use the Constant "srcKeyboard" as shown:

```
   @rs@ = IIF(srcKeyboard = ResponseSource(), "Last Response Keyed", "Not Keyed")
```

Using Functions in Scripts

Functions are small processing units that convert data from one form into another. ITScriptNet has a wide variety of functions available for In-Prompt Scripts. The Functions are grouped into categories to make accessing the functions in the Script Editor tree easier. Most functions take one or more parameters and return a string or numeric result. For example:

```
   Left("ABC123", 3)
```

would return “ABC”

Note that Logical operations in ITScriptNet are functions, not operators. For example, to test for "AND" you would use:

```
   And(True, False)
```

returns "FALSE".

There are well over a hundred Functions available. Some of the categories of functions include: String Functions, Math Functions, Conversion Functions, Notification Functions, Lookup Functions, and more. The Lookup Functions are especially important since they allow access in the scripts of both Validation Files data and the data collected in the program. For a full listing of the ITScriptNet functions with descriptions and examples, please refer to the Function Reference in this User Guide or the Script Editor screen’s Element Tree.

Using Keywords in Scripts

ITScriptNet has groups of Keywords that add the ability to loop and perform conditional execution of blocks of code. One group of Keywords includes the IF(<expression>), ELSE, ELSEIF(<expression>), and ENDIF. Multiple script lines can be in each segment of the IF block so that you can control the execution for many lines of code at once. An IF block must have one IF and a corresponding ENDIF and one or more statements that will execute if the expression is true. An IF block may also contain many ELSEIF segments as well as an ELSE segment. Two other groups of keywords provide the capability to have looping logic in your scripts so that a segment of script code can execute multiple times though a loop within an In-Prompt Script. These are FOR/NEXT and WHILE/WEND. One important rule for using keywords is that the keyword can not be nested with functions and the keyword must appear on its own line in the script with only its expression, if applicable. Please refer to the Function and Keyword Reference in this User Guide for more information about using Keywords.

Using Responses and Lookups in Scripts

You can access the data you collected in your scripts two ways:

Responses

The data collected by the operator is stored in Response variables. Use the "$" signs to delimit the Prompt name and Element name separated by a "." (period). For example, if you have a Prompt named "Prompt1" and a Text Input Element named "PartNo", you would reference the response to that Element with "$Prompt1.PartNo$". You may also assign a value to these variables in a script. This gives you the ability to override the collected data, if necessary.

Using Properties in Scripts

You can access the individual properties of an element in a script. The properties generally correspond to the options for the element that can be overriden by an in-prompt script. For example, Height, Width, BackgroundColor, Value, etc. Properties are referenced using the syntax

```
Prompt.Element.Property = value
```

For example, to set the width of an element named 'Textbox1' on the prompt named 'Prompt1' to 50 pixels, you would use:

```
Prompt1.Textbox1.Width = 50
```

You can also refer to the value of a property in your script, as follows:

```
@Width@ = Prompt1.Textbox1.Width
```

Using User Variables in Scripts

In addition to Response and Lookup variables, you may also create User-Defined variables. These are variables that you assign and use in your scripts. User-Defined variables are accessed by surrounding the name in "@". For example, "@UserVar@" is a user defined variable. The following characters may not be used in user variable names: # $ @ + - * . / = < > ( ) & or <space> as these characters are reserved. You do not have to declare these variables, but you may simply assign to them and evaluate them. They will be created as they are referenced. User defined variables keep their data as long as the data collection program is active. Once you escape back to the Main Menu these variables are no longer available. User defined variables may be used in scripts, in the Display Text in Single-Prompts, and in the Text Setting of Text Elements.

Variables without the @ delimiter are Local variables, which apply only to the script in which they are used. They go out of scope at the end of the script.

Using Operators in Scripts

The list of operators supported in ITScriptNet are shown in the table. You may use literals, functions, or variables on either side of the operator.

Addition	+
Subtraction	-
Multiplication	*
Division	/
Greater Than	>
Less Than	<
Greater Than or Equal	>=
Less Than or Equal	<=
Equal	=
Not Equal	<>
String Concatenation	&
Logical AND	&&
Logical OR	||

Data Types
Scripting in ITScriptNet is typeless. This means that all data is treated as strings except when numeric processing is performed, at which time the data will be converted into a numeric value. When converting from a string to a numeric value, the conversion stops at the first non-numeric characters. For example, “123.25AB” would be converted to 123.25. Some functions that require numeric parameters will display a syntax error if a non-numeric parameter is given.

Other In-Prompt Scripting Notes

How Scripts are Evaluated
Scripts allow nesting of functions and parameters. Scripts are always evaluated from the innermost parameter outward, and from left to right. You may nest functions as deeply as necessary. Scripts can assign data to variables. Each line of the script is executed sequentially and can either be an assignment or an evaluation. For example, the following is a valid In-Prompt Script:

```
@UserVar@ = Left($Descr$,10)
Beep()
```

This script would assign the left 10 characters of the data collected in the Prompt named "Descr"’ to the user-defined v

Conditional Branching
Using an In-Prompt Script for the Next Prompt field allows you to perform conditional branching. The results of the script should specify the name of the next Prompt to display. This allows you to control the program flow by changing the next Prompt. You can also use an In-Prompt Script to evaluate the Escape Prompt property. Other conditional branching and navigation options are available using the GoToPrompt function. Please refer to the Function and Keyword Reference in this User Guide for more information.

Saving Data
Data is saved when the last Prompt in the flowchart is accepted. This means that when using conditional branching, if the last Prompt is not processed, your data will not be saved!

If your program does not lend itself to a common last Prompt, there are three approaches to ensure your data is saved:

1. Add a Hidden (Skipped) Prompt as the last Prompt that all paths flow through. Even though the hidden Prompt is not shown, it will cause the data to be saved.
2. Add a confirmation or summary Prompt. You can add a Single-Prompt with a zero Max Length and Min Length, and it will be displayed but the only valid input will be to press ENTER or ESCAPE. You can display a summary of the collected information and allow the operator to press ENTER to save. You could also add a Multi-Prompt with summary information and a **Save** button.
3. Use the SaveCollectedData function to save the current value of all Prompts and Elements.

Note: You can also use the fact that the program saves on the last Prompt to create programs that do not collect data, but are used simply to perform lookups. In this case, simply have the second-to-last Prompt use an In-Prompt Script on the Next Prompt field (or the GoToPrompt function) to loop back to another Prompt. The last Prompt will never be displayed, and no data will be saved.

Using the Picklist Special Function

The Override Display Field In-Prompt Script is different from the others. This script is called once for every line in the validation file that is being used to fill a Combobox, Listbox, Listview or Gridview Element on a prompt. This function allows you to change the format of the string displayed in the list, or to exclude a record from appearing in the list. You use the PicklistField function to retrieve the value of a validation file field.

Comboboxes, Listboxes.

For these Prompt or Element types, the PicklistField function is used to format a string to be displayed in the list instead of the standard text that is composed of the Validation field and the Lookup Field. You can also skip records from the Validation file by returning an empty string from this script.

Listview and Gridview

For this Element type, you can override the data values in each grid cell in the text field override script for each field. To do this, set the value using the syntax $prompt.listview.field$ = “data value”.

Hidden (Skipped) Prompts

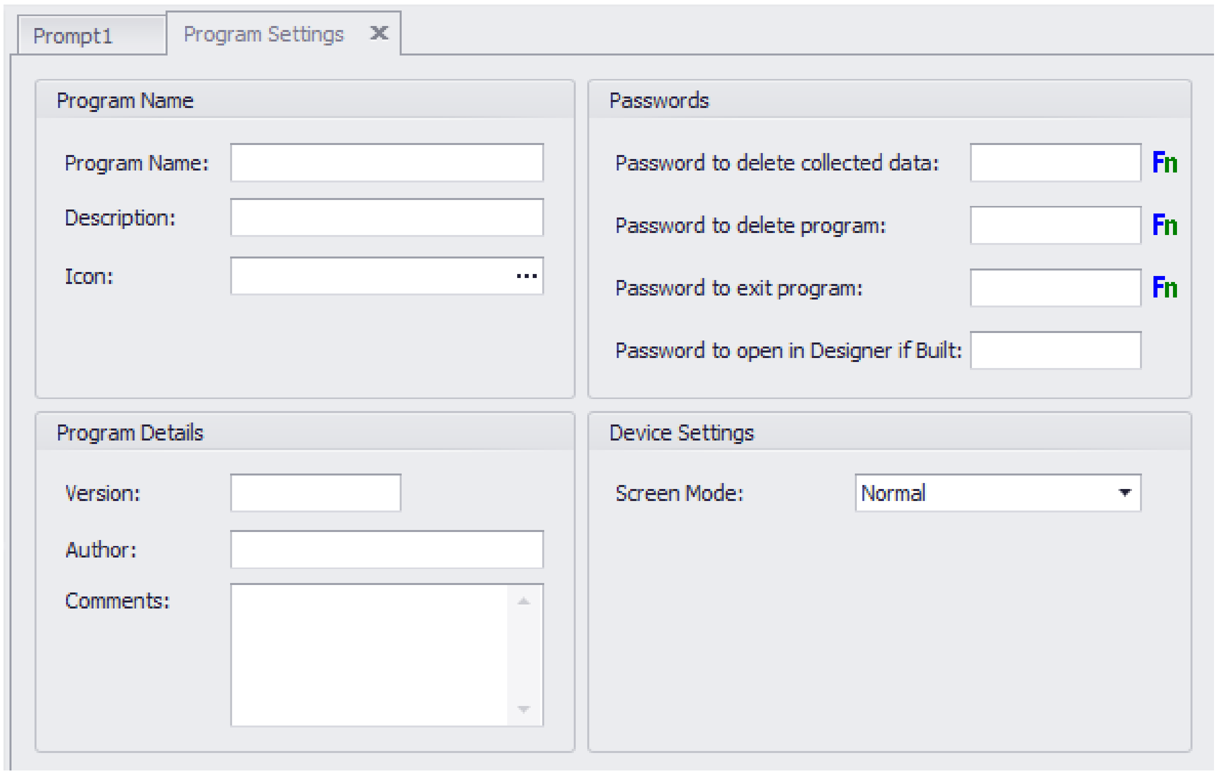
Only the Hidden Script is run if the Prompt is skipped.

Maximum Length

You may use an In-Prompt Script to control the maximum length property. However, the collected data will be truncated at the length you specified at design time, even if you adjust the maximum length to a larger value. If you need to adjust the maximum length, you should set the design-time value to the largest size you might need, and then set shorter limits at run-time in the in-prompt script.

3.2 Program Settings

The Prompt Settings screen has options that apply to the program as a whole. Some properties control how the program operates, and some are just for reference.



Program Name
The is the display name of the program in the Select Program list on the client. If you leave this blank, the Program Designer will default it to the ITBX file name. The text you enter here is available to the In-Prompt Scripts as the progITBName constant.

Description
This is the subtitle displayed in the Select Program list on the client, under the Program Name. IF you leave this blank, no description will be displayed. The text you enter here is available to the In-Prompt Scripts as the progITBDescr constant.

Icon
You can specify an Icon to be used in the Select Program list on the client. If you do not specify an icon, or if the icon file cannot be found on the device, then a default icon will be used. To ensure that the image you choose is available on the device, the Program Designer will add the image as a Support File so it gets loaded during the Load Program operation.

Version
This field allows you to enter a version number for the program. This is just a text field, and does not require any particular version format. The text you enter here is available to the In-Prompt Scripts as the progITBVersion constant.

Author

This is a reference field that you can use to record the author of the program. The text you enter here is available to the In-Prompt Scripts as the progITBAuthorconstant.

Comments
This is a reference field that you can use to record some comments about the program, such as the purpose of the program, or some notes about how the program is to be used. This field is not used by the program.

Password to Delete Collected Data
If you enter a password here, then the user of the device will be required to enter the password in order to delete collected data on the Utilities menu. You can execute an In-Prompt Script to determine this password at run time, but be aware that the program itself is not executing so you cannot reference any program elements. You could perform a Validation File lookup, however.

Password to Delete the Program
If you enter a password here, then the user of the device will be required to enter the password in order to delete the program from the Utilities menu. You can execute an In-Prompt Script to determine this password at run time, but be aware that the program itself is not executing so you cannot reference any program elements. You could perform a Validation File lookup, however.

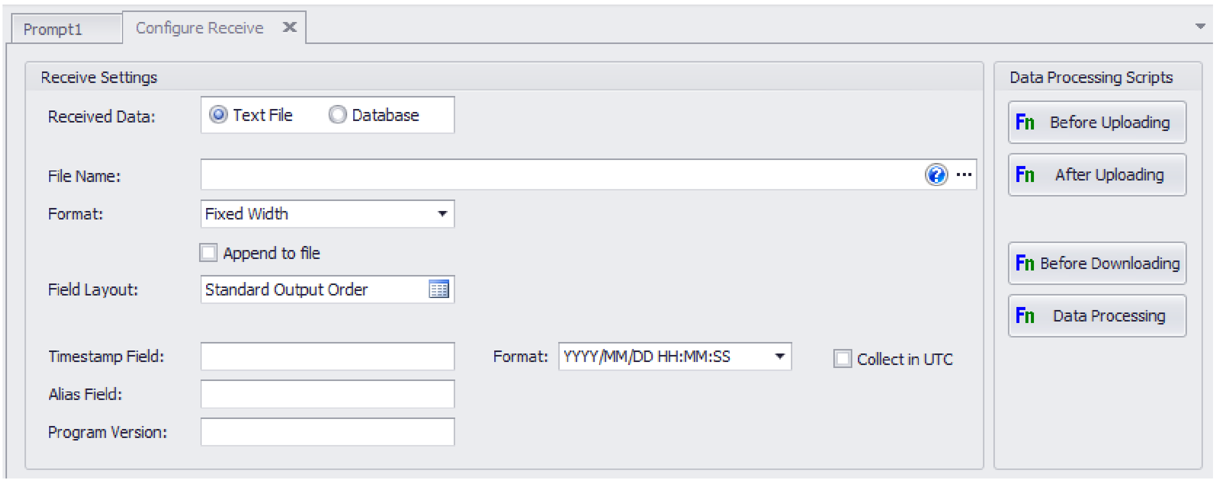
Password to Exit the Program
If you enter a password here, then the user of the device will be required to enter the password before they can exit the program. There is an In-Prompt script for this property, which does run in the Data Collection program.

Password to open on the Designer if Built
This password can be used to provide a measure of protection to prevent other people from opening your program and viewing the design. It only applies to programs Built into an ITBZ with the Build for Deployment option on the Utilities menu. When a user attempts to open the ITBZ with a program designer, they will be required to enter the password in order to recover the original ITBX.

Screen Mode
This setting controls the default Full Screen mode for the program on the device. You can override this setting per-layout, or use this setting for all prompts. Note that not all device types support all Full Screen modes. In general, the Full Screen and Normal modes are supported on almost all devices, while the Caption Only and Disabled Bars modes only apply to Windows CE/Windows Mobile.

3.3 Processing Collected Data

ITScriptNet provides powerful methods to process your collected data once it has been received by the OMNI Server.

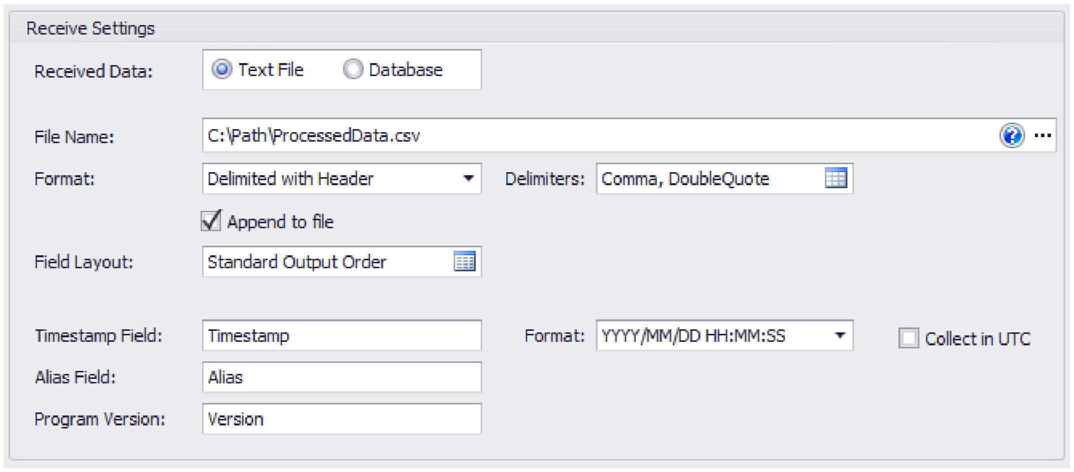


Received Data: (● Text File) ○ Database
File Name:
Format: Fixed Width
☐ Append to file
Field Layout: Standard Output Order
Timestamp Field:
Format: YYYY/MM/DD HH:MM:SS
☐ Collect in UTC
Alias Field:
Program Version:

The OMNI Server supports storing your collected data in a Text File or a Database. You can also write Data Processing scripts that allow you to modify the collected data before storing, or perform other tasks during the processing steps. The OMNI Server also supports a Deployment Override mechanism that makes it easy for you to override the Database Connection string, File path, or other settings that are different in a Production environment from your Development environment.

3.3.1 Processing to a Text File

The ITScriptNet OMNI Server can receive and store your collected data into a Text File. There are many options to control exactly how the data is stored.



Received Data:	[✓ Text File] Database
File Name:	C:\Path\ProcessedData.csv
Format:	Delimited with Header
Delimiters:	Comma, DoubleQuote
Append to file	✓
Field Layout:	Standard Output Order
Timestamp Field:	Timestamp
Format:	YYYY/MM/DD HH:MM:SS
Collect in UTC	☐
Alias Field:	Alias
Program Version:	Version

If you select the Text File method for your received data, you can configure these options.

File Name
This is the path to the file that will receive the collected data.

If you leave this blank, the OMNI Server will create a filename using the program file name, with a file extension depending on the Format. Fixed Width files will have a TXT extension, while Delimited files will have a CSV extension..

If you specify a fully-qualified path, the OMNI Server will use it to store the file. If you do not specify a path (just the filename), the server will store the file in the ITBX folder.

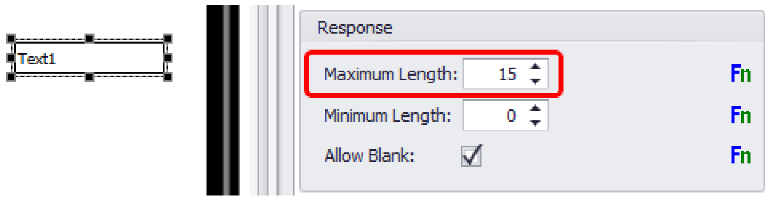
File Name Variables
You can use a few variables in your filename. When the Collected Data is downloaded and the data file is created, these variables will be translated as described below.

%DATE%	Replaced with the current Date, in the format YYYYMMDD (ex: 20130630)
%TIME%	Replaced with the current Time, in the format HHMMSS (ex: 142356)
%YEAR%	Replaced with the current Year (ex: 2013)
%MONTH%	Replaced with the current Month (ex: 12)
%DAY%	Replaced with the current Day of the Month (ex: 30)
%HOUR%	Replaced with the current Hour in 24-hour time (ex: 14)
%MINUTE%	Replaced with the current Minute (ex: 58)
%SECOND%	Replaced with the current Second (ex: 30)
%MACHINENAME%	Replaced with the Machine Name of the PC receiving the data (ex: SERVER1)
%PROGRAMNAME%	Replaced with the Program Name, which is the name of the ITBX file without the file extension
%UNIQUEID%	Replaced with a GUID to yield a unique ID (ex: 89F6C9E7-C973-4CA0-A850-1774DA260945)

Format

This setting controls the format of the output text file. You can choose from:

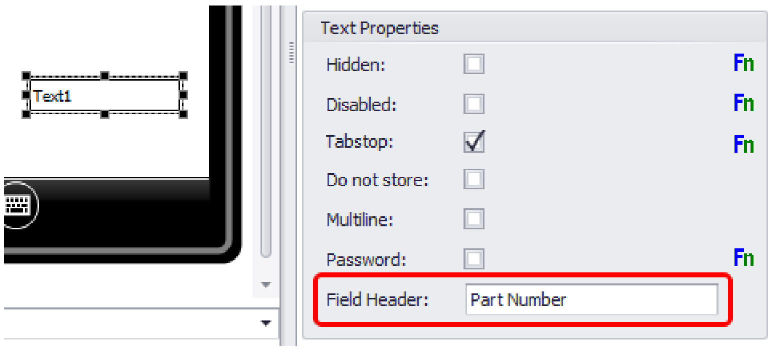
Fixed Width: The output file is a fixed width file with no headers. Each field will be space padded to the width you specified for the element.



Maximum Length: 15
Minimum Length: 0
Allow Blank: ✓

Delimited without Header: The output file is a delimited file with no headers. The fields are not space padded. The Text Delimiters and Qualifiers you choose are used to separate the fields.

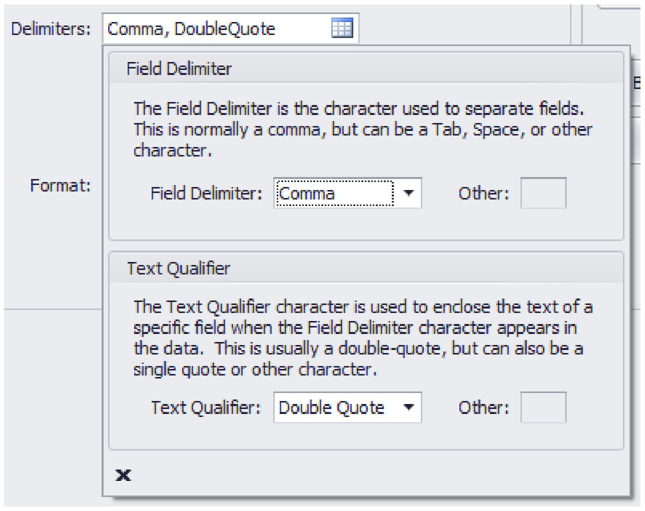
Delimited with Header: The output file is a delimited file with headers. The fields are not space padded. The Text Delimiters and Qualifiers you choose are used to separate the fields. The field headers will be written to the file if the file is created, but not if the file already exists. The Field Headers are the values provided when you defined the elements.



Hidden: ☐
Disabled: ☐
Tabstop: ✓
Do not store: ☐
Multiline: ☐
Password: ☐
Field Header: Part Number

Delimiters

If you selected a delimited file type, this area is where you choose what the delimiters should be. The default delimiter is a comma, and the default qualifier is double-quote.



Delimiters: Comma, DoubleQuote

The Field Delimiter is the character used to separate fields. This is normally a comma, but can be a Tab, Space, or other character.

Field Delimiter: Comma Other:

The Text Qualifier character is used to enclose the text of a specific field when the Field Delimiter character appears in the data. This is usually a double-quote, but can also be a single quote or other character.

Text Qualifier: Double Quote Other:

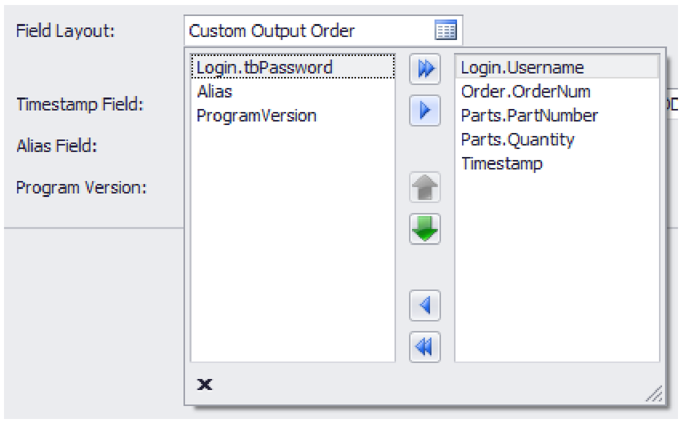
You can selected from a set of predefined Delimiters and Text Qualifiers, or select 'Other' and enter your own character.

Append To File

This setting controls whether a new file is created on each download (overwriting an existing file with the same name), or if the new data should be appended to the file if it already exists.

Field Layout

This section allows you to control the order that the fields are written to the file. By default, the fields are written in the order that they appear in the Tab Order on each prompt. However, if your requirements are to have the fields in a specific order,


Field Layout: Custom Output Order

Timestamp Field:

Alias Field:

Program Version:

Login.tbPassword
Alias
ProgramVersion

Login.Username
Order.OrderNum
Parts.PartNumber
Parts.Quantity
Timestamp

The fields on the left side are not used, and the fields on the right side will be output in the order from top to bottom. You can move the fields from left to right, or right to left using the blue arrows. You change the order by moving the fields on the right side up and down.

Timestamp Field

Every record collected on the device is marked with a timestamp indicating the time it was collected. This field allows you to specify the file header text for this collected timestamp. You can select from a number of predefined Timestamp formats. You can also select to have the timestamp collected and output in UTC instead of the device's local time.

Alias Field

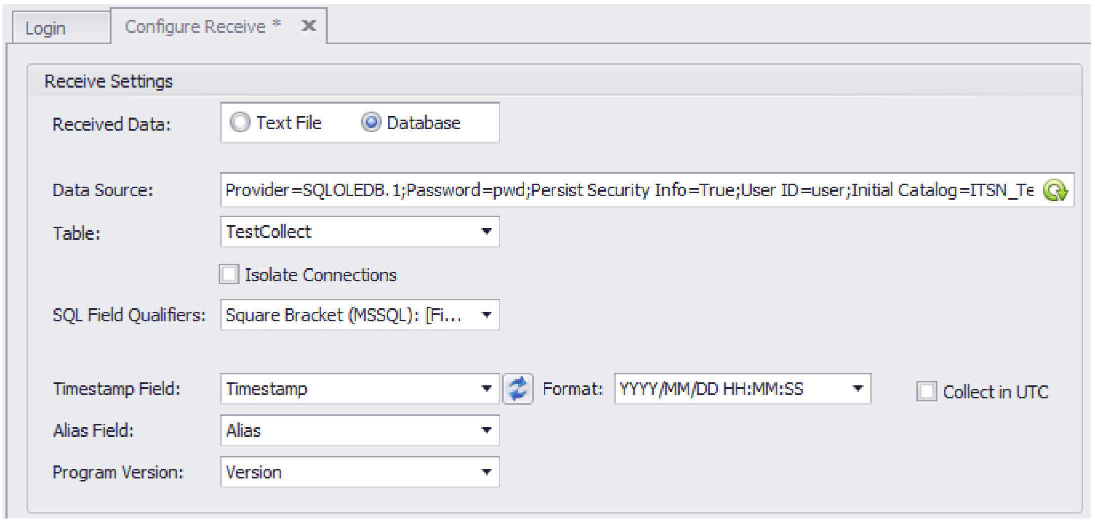
Every record collected on the device is marked with the Alias of the device. This field allows you to specify the file header text for the Alias field.

Program Version

Every record collected on the device is marked with the version of the ITBX. This field allows you to specify the file header text for the Program Version field.

3.3.2 Processing to a Database

The ITScriptNet OMNI Server can receive and store your collected data into a Database. There are many options to control exactly how the data is stored.



Received Data: ( ) Text File [✓ Database]

Data Source: Provider=SQLOLEDB.1;Password=pwd;Persist Security Info=True;User ID=user;Initial Catalog=ITSN_Te

Table: TestCollect

☐ Isolate Connections

SQL Field Qualifiers: Square Bracket (MSSQL): [Fi...

Timestamp Field: Timestamp Format: YYYY/MM/DD HH:MM:SS ☐ Collect in UTC

Alias Field: Alias

Program Version: Version

Data Source
This field specifies the database connection. There are three connection types supported.

ODBC

This option uses an ODBC Data Source to connect to the database. If the connection string starts with DSN= then it is considered an ODBC connection.

Ex:
  DSN=DataSourceName

Note that ODBC Data Sources are either 32-bit or 64-bit. A 32-bit application can only access 32-bit DSNs, and a 64-bit application can only access 64-bit DSNs. They cannot be mixed. Some ODBC drivers are only available in 32-bit or 64-bit. This means you must select the correct runtime to match the ODBC drivers your database provides.

Press the ODBC button to bring up the list of Data Sources to choose from.

OLEDB

This option uses an OLEDB Connection String to connect to the database. OLEDB is a more modern method than ODBC, but there are still some databases that do not have OLEDB drivers. If the connection string starts with Provider= then it is considered an OLEDB connection.

Ex:
  Provider=SQLOLEDB.1;Password=pwd;Persist Security Info=True;User ID=user;Initial Catalog=Database;Data Source=Server

Press the OLEDB button to bring up the Data Link Editor, which assist you in connecting to the database and building the connection string.

MSSQL

This option makes a direct connection to a Microsoft SQL Server database. If the connection string does not start with DSN= or Provider= then it is considered an MSSQL connection string.

Ex:
  Data Source=Server;Connection Timeout=60;Initial Catalog=Database;Persist Security Info=True;User ID=user;Password=pwd

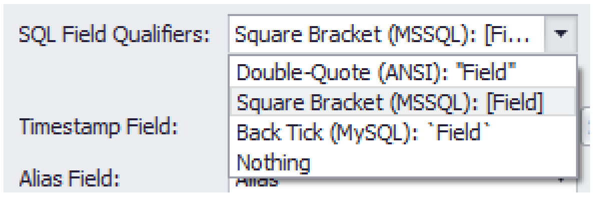
Press the MSSQL button to bring up the SQL Server connection dialog, which assist you in connecting to the database.

Table
This field holds the table where the collected data should be stored. Once you have set the connection string, this field will be filled from the list of all tables available in the database.

Isolate Connections
If this option is set, then all database connections will be isolated so that only one connection is made at a time. If two or more devices have data to be processed at the same time, then each connection will be made one at a time while the others wait.

This option is not generally needed, but is available for database which do not support multiple simultaneous connections.

SQL Field Qualifiers
Each database engine can have a its own syntax for delimiting fields in a SQL query. This selection lets you choose the field qualifiers to use when the OMNI Server generates the SQL Statements used for inserting the collected data into the selected table.


SQL Field Qualifiers:
Square Bracket (MSSQL): [Fi...
Double-Quote (ANSI): "Field"
Square Bracket (MSSQL): [Field]
Back Tick (MySQL): `Field`
Nothing
Timestamp Field:
Alias Field:

The options include:
- Double-quotes - This is the ANSI SQL standard option, used by many database engines.
- Square Brackets - This is the default used my Microsoft SQL Server.
- Back Tick - This is the default used by MySQL,
- Nothing - no field qualifier will be used.

Set the option to best match the database engine you are using. If this option is set to a qualifier that your database engine does not support, you will see a SQL error in the OMNI Server log file after attempting to process collected data.

Timestamp Field
Every record collected on the device is marked with a timestamp indicating the time it was collected. This field allows you to specify the database field to receive the collected timestamp.

You can select from a number of predefined Timestamp formats. If the underlying database field is a Text type, the timestamps will be stored in the text format specified. If the underlying database field is a Date/Time type, then the database will attempt to convert the text representation of the timestamp into a Date/Time field. Since difference database engines support different parsing of Date/Time strings, you will have to select a format that matches your database engine.

Most database engines can support the YYYY/MM/DD HH:MM:SS format.

You can also select to have the timestamp collected and output in UTC instead of the device's local time.

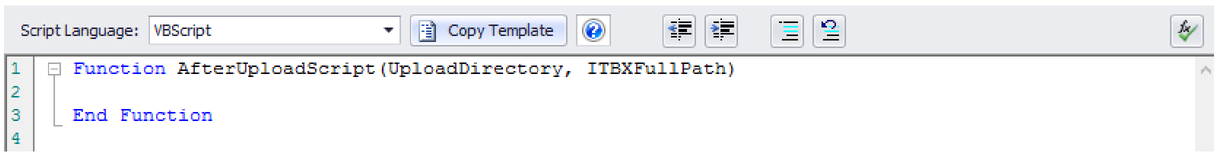
Alias Field
Every record collected on the device is marked with the Alias of the device. This field allows you to specify the database field to receive the collected Alias.

Program Version
Every record collected on the device is marked with the version of the ITBX. This field allows you to specify the database field to receive the collected Program Version.

3.3.3 Data Processing Scripts

There are several Data Processing scripts available. These are scripts that run on the OMNI Server at various points during the communication process and allow you to perform additional processing for your application. Unlike the In-Prompt scripts which run on the device and use the ITScriptNet script language, these Data Processing scripts can be written in C#, VB.Net, VBScript, or JavaScript.

Data Processing Script Editor
All of the data processing scripts use the same basic editor.



```
Function AfterUploadScript(UploadDirectory, ITBXFullPath)

End Function
```

You can select the language to be used by the script. This tells the OMNI Server which script engine should be used to execute the script.

The Copy Template button initializes the script to an empty template, with the correct declarations and definitions so the script will execute. All you have to do is fill in your custom processing.

There are Indent/Outdent buttons as well as Comment/Uncomment buttons in the tool bar.

The Fx button performs a syntax check on the script. This check can find basic errors in the syntax of your program.

Before Upload
This script is executed by the OMNI Server during the Load Program process. It happens after the OMNI Server has created the upload temporary directory, but before any files are copied to it or any validation files generated. You could use this script to pull data from an external data source, or copy a flat file, or any other operation that must be done before each Load Program.

The UploadDirectory parameter is the temporary folder that the OMNI Server creates to hold all of the files that will be sent to the device.

The ITBXFullPath parameter is the fully qualified path to the ITBX that is being loaded.

After Upload
This script is executed by the OMNI Server during the Load Program process. It happens after all files have been sent to the device, and just before the temporary directory is deleted. You could use this script to perform any clean-up tasks that you need after a program load.

The UploadDirectory parameter is the temporary folder that the OMNI Server creates to hold all of the files that will be sent to the device.

The ITBXFullPath parameter is the fully qualified path to the ITBX that is being loaded.

Before Download
This script is executed by the OMNI Server during the processing of collected data. It happens after the temporary download directory has been created, but before the files are received from the device. You could use this script to perform any processing that you need to do before processing the collected data.

The DownloadDirectory parameter is the temporary folder that the OMNI Server creates to receive all of the files from the device.

The ITBXFullPath parameter is the fully qualified path to the ITBX that is being processed.

Data Processing
These scripts are executed as the collected data is being processed. There are 3 separate functions that will be called.

Before Processing
This script is called after the data has been downloaded, but before any record have been processed.

The DataDirectory is the temporary folder that the OMNI Server creates to receive all of the files from the device.

The ITBXFullPath parameter is the fully qualified path to the ITBX that is being processed.

Every Record
This script is called once for each record in the collected data.

The DataDirectory is the temporary folder that the OMNI Server creates to receive all of the files from the device.

The ITBXFullPath parameter is the fully qualified path to the ITBX that is being processed.

After Processing
This script is called once at the end, after all records have been processed.

The DataDirectory is the temporary folder that the OMNI Server creates to receive all of the files from the device.

The ITBXFullPath parameter is the fully qualified path to the ITBX that is being processed.

Accessing the Collected Data from the Scripts
The collected data is available to the "On Every Record" script in a pre-defined collection named ResponseList, keyed by the Prompt Name for single Prompts and by “Prompt.Element” for Multi-Prompts. For example, if you have a Prompt named “Prompt2” that has an Element named “ElementA”, you can access the data collected for that element by using ResponseList("Prompt2.ElementA"). You may make changes to the collected data and save those changes back to the ResponseList. Changes made in this way will be saved in the collected data file. Any collected data fields that are not modified will be saved as they were collected.

Note that the data will only be populated in the ResponseList in the "On Every Record" script.

Passing Objects Between Scripts
You can pass objects (such as Database Connections, Recordsets, or COM Objects) between the scripts using the predefined collection named ObjList. This collection is keyed by a name you specify. For example, you could store a database connection as ObjList("Connection"). The ObjList retains the objects placed into it from one script to the next. See the Script Sample below for an example.

Note that the ObjList is only available in the "Before Processing" Data, "On Every Record", and "After Processing" Data Scripts.

Skipping Records
You can force a record to be skipped and not placed in the collected data file. To do this, set the predefined variable named ProcessRecord = 0. This will cause the data record to be skipped as though it had not been collected.

Sample Script
The following is a sample Data Processing script in VBScript:

```
' Do not remove these declarations
Set ResponseList = CreateObject("Scripting.Dictionary")
ResponseList.CompareMode = vbTextCompare
Set ObjList = CreateObject("Scripting.Dictionary")
ObjList.CompareMode = vbTextCompare
ProcessRecord = 1

' This function is executed before the collected data records are
processed
Function BeforeProcessingScript(DataDirectory, ITBXFullPath)
   Set adoConn = CreateObject("ADODB.Connection")
   adoConn.Open "DSN=Function"
   Set adoRS = CreateObject("ADODB.Recordset")
   adoRS.Open "select * from function where ID = -1", adoConn, 3, 3
   Set ObjList("Conn") = adoConn
   Set ObjList("RS") = adoRS
End Function

' This function is executed once for every collected record
Function OnEveryRecordScript(DataDirectory, ITBXFullPath)
   Set adoRS = ObjList("RS")
   adoRS.AddNew
   adoRS.Fields("PrePrompt") = ResponseList("preprompt")
   strDate = trim(ResponseList("timestamp"))
   adoRS.Fields("RealDate") = mid(strDate, 5, 2) & "/" & mid(strDate, 7,
2) & "/" & left(strDate, 4) & ResponseList("Alias") = "NewTerm"
   adoRS.Update
End Function

' This function is executed after the collected data records are
processed
Function AfterProcessingScript(DataDirectory, ITBXFullPath)
   Set adoConn = ObjList("Conn")
   Set adoRS = ObjList("RS")
   adoRS.Close
   adoConn.Close
   Set adoRS = Nothing
   Set adoConn = Nothing
End Function
```

Before Processing Script:
The Script creates and opens a database connection and an empty recordset, then stores them in the ObjList so they will be available to the other scripts.

On Every Record Script:
This Script is run once for each record. The recordset is retrieved from the ObjList, then the data for the fields is read from the ResponseList. In this sample, the timestamp field is reformatted and the alias is changed. Note that you can change the data and assign it back to the ResponseList. The changed data will be processed as though the device operator had collected it. The other fields in ResponseList are saved unchanged.

After Processing Script:
This script cleans up the objects by closing the recordset and database connection.

3.3.4 Deployment Override

Many Validation File and Collected Data parameters can be overriden for deployment. The mechanism for this is an INI File with the same filename as the ITBX file, and located in the same directory.
For example, if your program file is called C:\Folder\CollectData.ITBX, then the INI file would be called C:\Folder\CollectData.INI.

To aid in creating your override INI file, there is an option in the System Console that loads the ITBX file, presents all of the possible options available, and saves the INI file. See the Runtime User's Guide for the details on this utility.

Any setting which is left blank will use the setting you used in the designer.

Master Connection
This is an override for all connection strings used by the program. If this is set, then all connection strings (GPS, Download, and all Validation Files) will use this setting unless they have a specific setting of their own.

3.4 Prompts

The Prompt contains any number of elements that are used to collect data. The Prompt settings control how the prompt is displayed and how to responds to user input.

Next Prompt
This setting determines the prompt that will be activated if the current prompt is Accepted. Generally, this is the next prompt in the flowchart. You can use the Property Script to override the Next Prompt at run time. This script is evaluated when the prompt is Accepted rather than when the prompt is initialized. This is also the only script evaluated if the prompt is Skipped.

Escape Prompt
This setting determines the prompt that will be activate if the current prompt is Canceled. You can use the Property Script to override the Escape Prompt at run time. This script is evaluated when the prompt is Canceled rather than when the prompt is initialized.

Force Rotation
This setting controls screen rotation for device that support rotation. You can lock the rotation to Portrait or Landscape, or allow the rotation based on device orientation.

Do No Prompt (Skip)
This setting causes the execution of the program to skip this prompt. If this is set, only the Next Prompt script is executed.

Do Not Save Data for this Prompt
This setting controls whether the prompt saves to collected data. If set, no data from the prompt will be written to collected data.

Prevent Exit on Escape/Exit
This setting can be used to prevent the user from backing out of the prompt by using the Escape key or by pressing a button with the Action set to Exit.

Tab Order
The Tab Order is the order that the keyboard focus automatically moves. The focus moves automatically when one of these happens:

1) The Tab key is pressed
2) The Enter is pressed and the After Enter action is set to Next Element
3) A barcode is scanned and the After Scan action is set to Next Element.

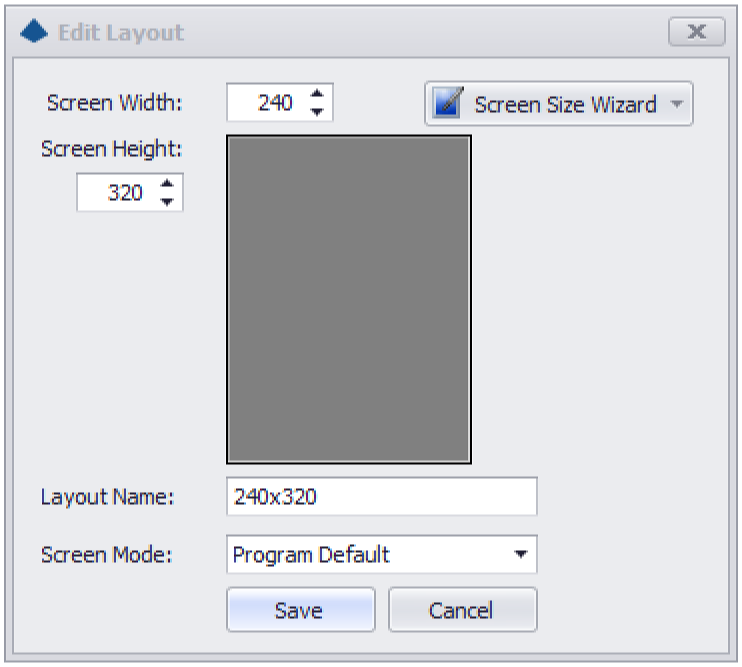
Most elements have a property called Tabstop that determines whether the element is in the Tab Order. If this property is not set, then the element is not in the Tab Order.

You can control the Tab Order on the Prompt Settings screen.

3.4.1 Prompt Layouts

Every prompt has one or more layouts that determine where elements are positioned. You can create layouts for various screen sizes and orientations, and the client will select the appropriate layout to match the device screen size.

Editing Layouts
You can create as many layouts as you need to support your devices.



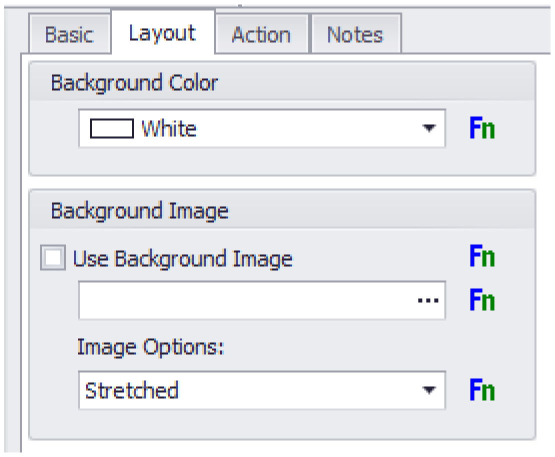
Screen Width: 240
Screen Height: 320
Layout Name: 240x320
Screen Mode: Program Default

The screen width and height should match your device screen size.

The Layout Name will automatically update to match, but you can override the name if you want.

The Screen Mode controls the Fullscreen mode for this prompt, and can override the Fullscreen mode set in Program Settings.

Prompt Layout Settings
This section of the Prompt Properties controls layout properties such as the background color or background image.



Background Color

White

Background Image

Use Background Image

Image Options:

Stretched

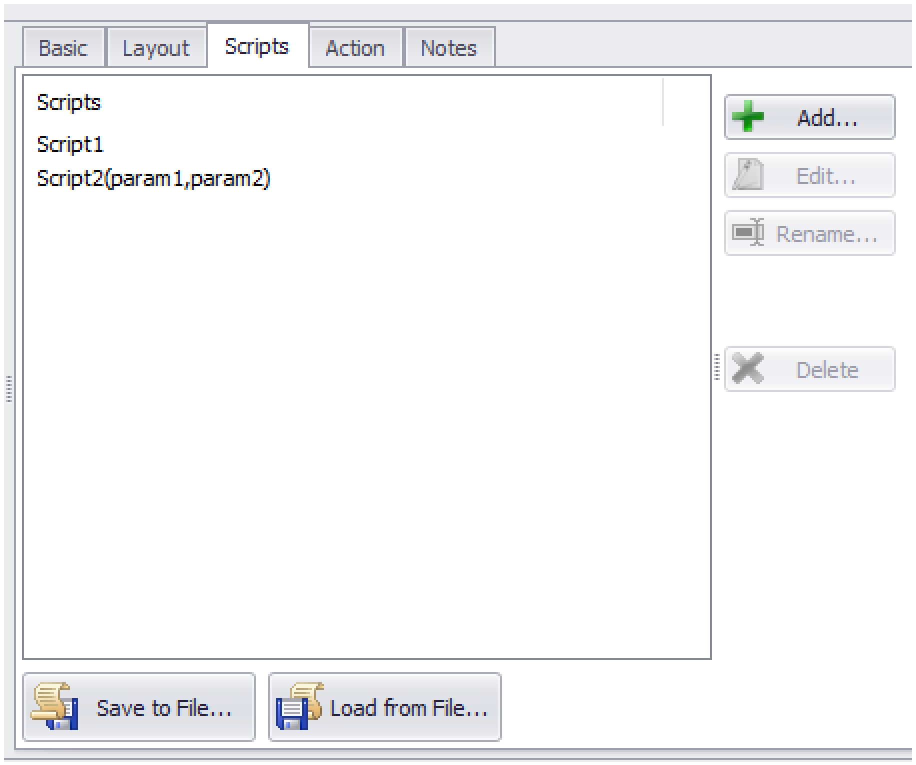
Layout Selection
The client will select a layout that best matches the screen size. This match can be an exact match, or an exact multiple of the layout size to the screen size. If there is no exact match, the closest match will be selected.

Overriding the Layout Selection
The client attempts to select the best layout match, but you can override this at run time. You can return the name of a Layout from the Before Layout Changed script to override the automatic layout selection. If you do not return a name, or if the name you return does not match a layout, then the automatically selected layout will be used.

3.4.2 Prompt Scripts

Each prompt can have scripts associated with it. These scripts can be called from any element script or event on the prompt using the PromptScript function.

Prompt scripts can only be called from the prompt where they were defined. If you need a script that can be called from anywhere in the program, use a Global Script.



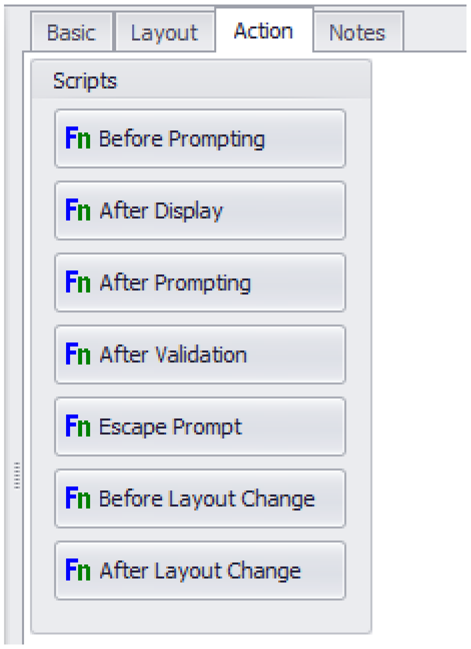
Scripts
Script1
Script2(param1,param2)

Press Add to create a new Prompt Script. The parameters you enter in the script name will be available as Local Variables in the script.

You can use the Edit button to edit the script itself, and the Rename button to change the name and parameters of the script.

3.4.3 Prompt Lifecycle

This topic discusses the events that occur during the lifetime of a data collection prompt. The scripts listed on the prompt's Action tab are called at various times as the prompt is loaded, unloaded, and during data collection.



Prompt Initialization

The following events occur when a prompt is begin created

Before Prompting
The first event in the prompt lifecycle is Before Prompting. This event is called once the prompt is loaded but before any elements are created. For this reason, you cannot reference element values in this script. For example, you cannot set a value to an element in this event, as the value will be overridden once the elements are created.

Element Creation
During this phase of the prompt lifecycle, the individual elements are created. The individual element property scripts are evaluated during this time.

Before Layout Changed
This event is called just before the layout is selected. You can return the name of a Layout from this script to override the automatic layout selection. If you do not return a name, or if the name you return does not match a layout, then the automatically selected layout will be used.

After Layout Changed
This event is called after the layout is selected.

After Display
This event is called after all elements have been created and initialized. It is the last event called before data collection starts. Since the elements have all been created, you can use this event to override element settings, including size, position, and value.

Data Collection

This is the phase where the operator interacts with the device and fills in element values.

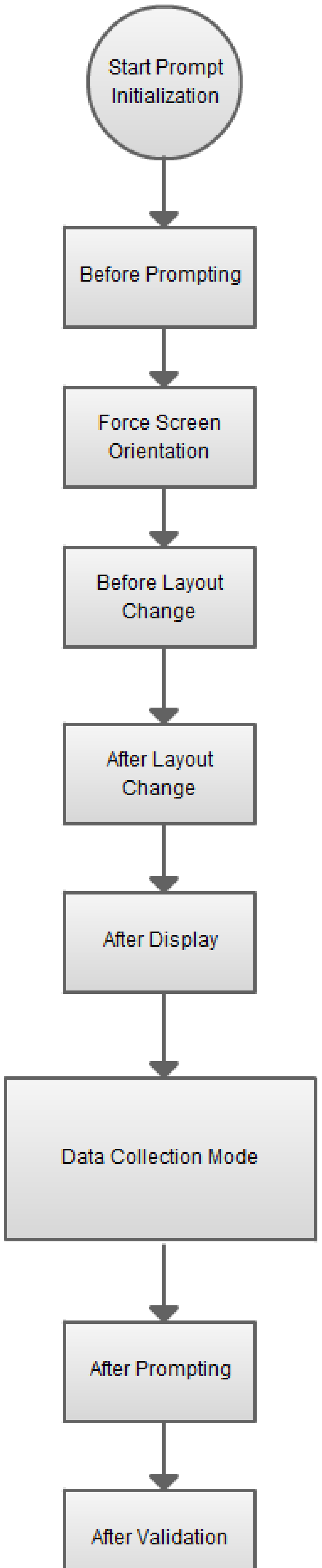
Before Layout Changed
If the device is rotated, this event is called just before the layout is selected.

After Layout Changed
This event is called after the layout is selected.

Prompt Accepted

These events occur after the prompt is accepted.

After Prompting
When the prompt is Accepted, either by a call to AcceptPrompt() or by the user pressing a button with the Action set to Accept Prompt, this event is called. This event occurs before any individual element



validations are called. If you call ValidationFail() in this event script, the Accept is canceled and the current prompt stays active.

Element Validations
The individual element validations occur during this phase. If any element fails its validation, or if ValdationFail() is called in the elements Validation script, then the Accept is canceled and the current prompt stays active.

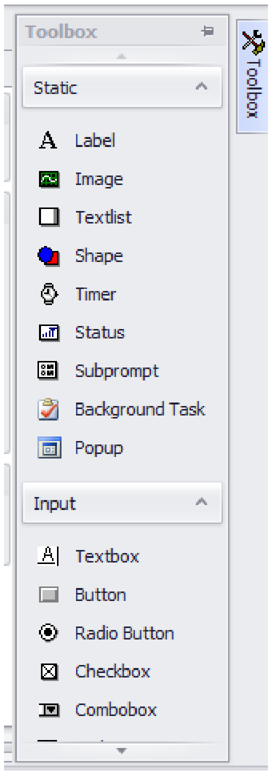
After Validation
This event script is called after all element validations have been executed. This is the last event that occurs before the prompt is destroyed and execution transferred to the next prompt. If you call ValidationFail() in this script, then the Accept is canceled and the current prompt stays active.

Prompt Canceled

Escape Prompt
This event is called if the user cancels the prompt, either by pressing a button with the Action set to Cancel, or by calling CancelPrompt, or by pressing Escape on devices with a keyboard. If you call ValidationFail() in this script, then the prompt will not be canceled and the current prompt stays active.

3.5 Elements

An Element is a control that collects a piece of data, displays a piece of data, or performs an action. Most elements are a visible control that the user can interact with, but some elements are invisible.



You add an element to a prompt using the Toolbox. Click one of the elements to create an add that element to the prompt.

The element properties will be displayed to the right of the prompt. You can adjust any of these properties to control how the element behaves.

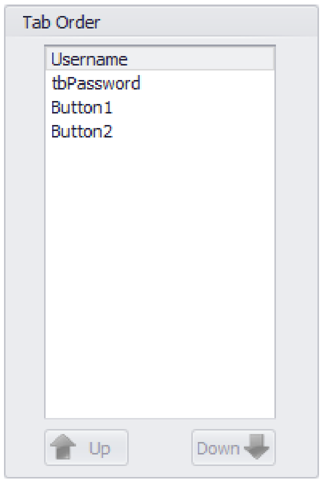
Elements are divided into two types: Static and Input.

Static elements do not collect any data, but are used to display information or are used to perform some action without user interaction. These include static text labels, images, shapes, timers, etc.

Input elements collect a piece of data and save it to the Collected Data. These include text boxes, listviews, buttons, etc.

Focus and Tab Order

The term Focus refers to the element that the user is currently interacting with. On devices with keyboards, the Tab key advances the focus from element to element. Pressing Enter or scanning a barcode can also advance the focus, depending on the After Enter and After Scan properties that most input elements support. On devices without keyboards, pressing Enter, Scanning, or tapping an element is the only way to move the focus.



Focus is usually indicated visually by either an outline rectangle or a color change. This is also configurable in the element properties.

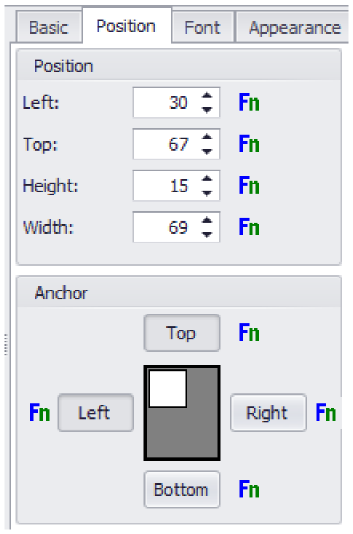
The Tab Order is set on the Prompt Properties screen. The focus moves in the order specified.

You can also use the Tabstop property to indicate that an element should not appear in the Tab Order. An element with the Tabstop property cleared will not receive the focus by pressing tab or enter on other elements. It will receive the focus if the element itself is tapped.

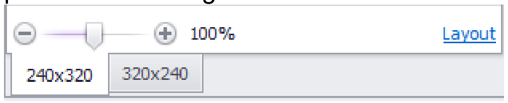
3.5.1 Position, Fonts and Layouts

Every element that is visible to the user has a size and position that depends on the prompt Layout. This means that the element can be positioned differently depending on whether the device is portrait or landscape, for example.

Positon



When you select the Position tab for an element, the Position and Anchor for the currently selected layout will be displayed. You will notice that if you change layouts, the changes you make to the size and position will change as well.



In addition to the position, you can also use Anchoring to control how the element is positioned. Anchoring controls what happens if the size of the device screen is not exactly the same as the size of the layout. For example, if you design the program for a Windows Mobile standard screen size of 240x320 but load the program on an Android device with a screen size that is a multiple of 240x400 (480x800, for example), there will be extra space at the bottom of the screen.

Anchoring works on Left/Right and Top/Bottom independently.

Left/Right

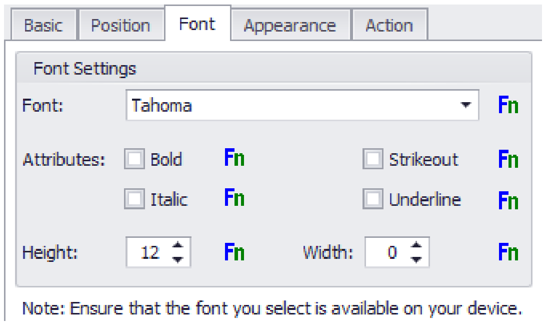
Left	Right	Result
Yes	Yes	Element stretches to keep the same left/right margins to the edge of the screen.
Yes	No	Element maintains its position to the left edge of the screen, and the size does not change.
No	Yes	Element maintains its position to the right edge of the screen, and the size does not change.
No	No	Element is centered horizontally and the size does not change.

Top/Bottom

Top	Bottom	Result
Yes	Yes	Element stretches to keep the same top/bottom margins to the edge of the screen.
Yes	No	Element maintains its position to the top edge of the screen, and the size does not change.

No	Yes	Element maintains its position to the bottom edge of the screen, and the size does not change.
No	No	Element is centered vertically and the size does not change.

Fonts
Fonts are also set on a per-layout, so you can display at different sizes for different layouts.



Font: Tahoma
Attributes: Bold, Italic, Strikeout, Underline
Height: 12
Width: 0
Note: Ensure that the font you select is available on your device.

The fonts that are available vary from device to device. The Program Designer always shows the TrueType fonts installed on your PC, but these fonts may not be installed on your device. Windows Mobile/Windows CE and Android device can use a TrueType font that is copied to the device along with the Data Collection program. For more device-specific details, see the Font topic in the Device Specific Notes section.

3.5.2 Label

The Label element displays some text on a prompt. You can control the text font and color, as well set add a drop shadow.

Changing the text at Run Time
There are several ways to change the text of a label while the program is running.

<u>Property Script</u>
There is a property script for the text property. If this script is used, the label will display the text returned as the result of the script. This script is executed with the prompt is initially loaded, or when the Refresh function is called on the element.

<u>Value</u>
You can set the Value of the element in a script, using the syntax

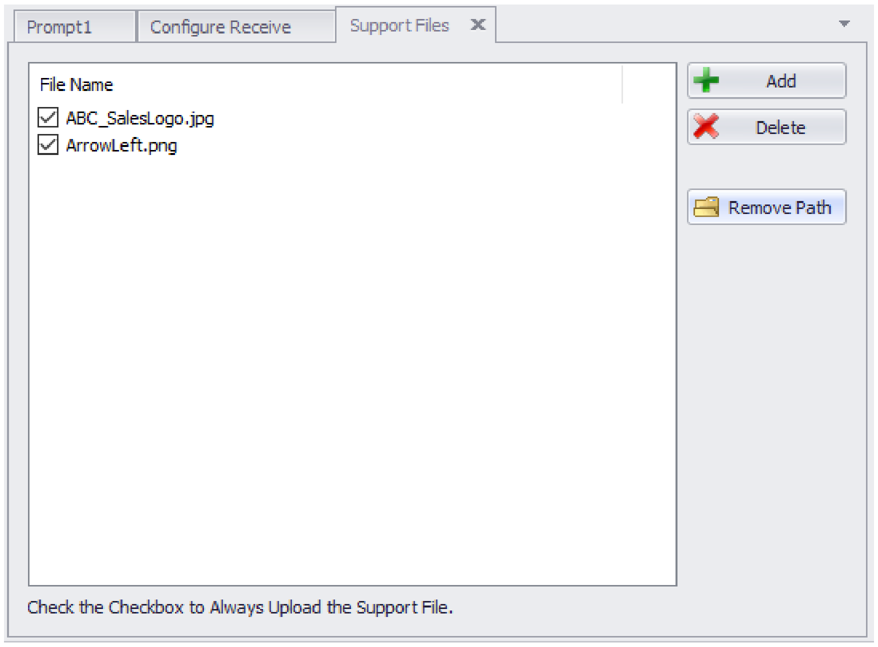
```
$Promptname.Elementname$ = "The text to display".
```

<u>Property</u>
You can set the value of the element in a script, using the Property syntax

```
Promptname.Elementname.Text = "This text to display".
```

3.6 Support Files

When the data collection program is loaded to the device, it is usually necessary to load any image files or other required files that the program uses. To simplify this process, you can assign files as Support files, and these files will get loaded during the Load Program process.



File Name
☑ ABC_SalesLogo.jpg
☑ ArrowLeft.png

Check the Checkbox to Always Upload the Support File.

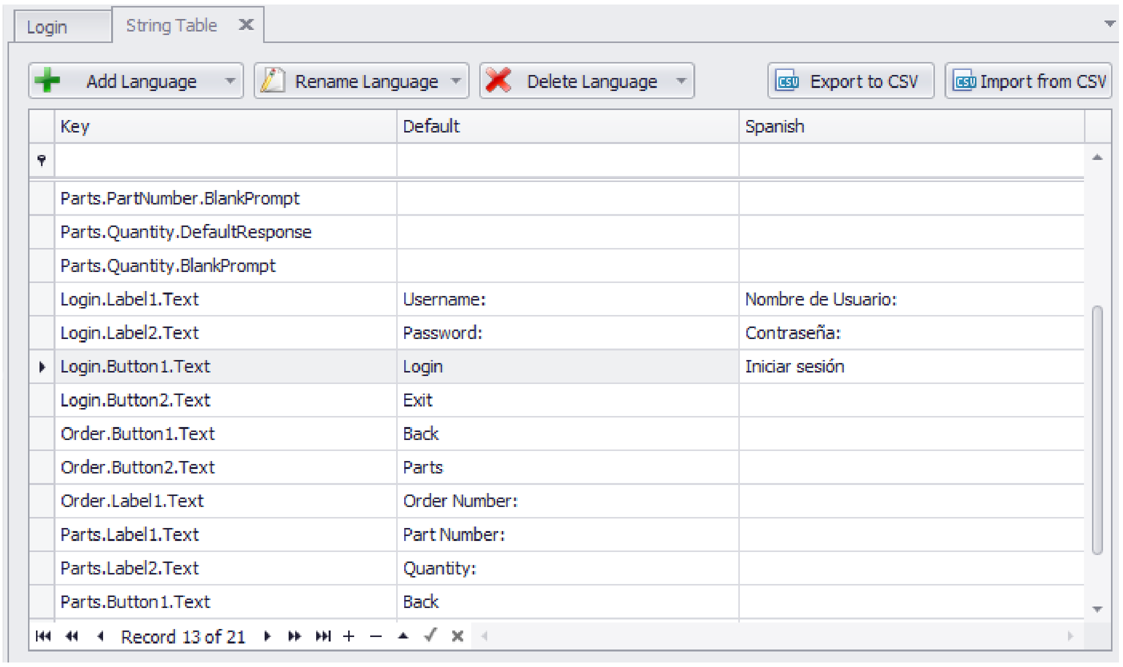
Generally, when you add an image to a button or other element, the designer will automatically add the image to the support files list for you.

By default, the items will be added with the checkbox checked. This checkbox determines whether the file will always be uploaded to the device or not. When loading the program from the Load Program menu on the device, all files are always loaded. However, when calling the OmniLoadProgram function from within an In-Prompt Script, there is a parameter that controls whether All Files are loaded or not. You might use this if there is an extremely large file that you do not always need to load or that changes infrequently. By unchecking the checkbox for that file on the Support Files screen and then calling OmniLoadProgram with the All Files parameter 0, the file will not be loaded.

Use the Remove Path option if the support file was added with a fully qualified path but you want to move the files to a different folder. If there is no path specified, then the file is assumed to be in the same directory as the ITBX.

3.7 String Tables for Language Support

ITScriptNet Indago has a mechanism that you can use to localize your program for different languages. The String Table is the basis of this support.



Key	Default	Spanish
Parts.PartNumber.BlankPrompt		
Parts.Quantity.DefaultResponse		
Parts.Quantity.BlankPrompt		
Login.Label1.Text	Username:	Nombre de Usuario:
Login.Label2.Text	Password:	Contraseña:
Login.Button1.Text	Login	Iniciar sesión
Login.Button2.Text	Exit	
Order.Button1.Text	Back	
Order.Button2.Text	Parts	
Order.Label1.Text	Order Number:	
Parts.Label1.Text	Part Number:	
Parts.Label2.Text	Quantity:	
Parts.Button1.Text	Back	

Whenever you add an element to a prompt, a corresponding entry will be added to the String Table, with a column for each defined language. You can add and edit languages with the controls at the top.

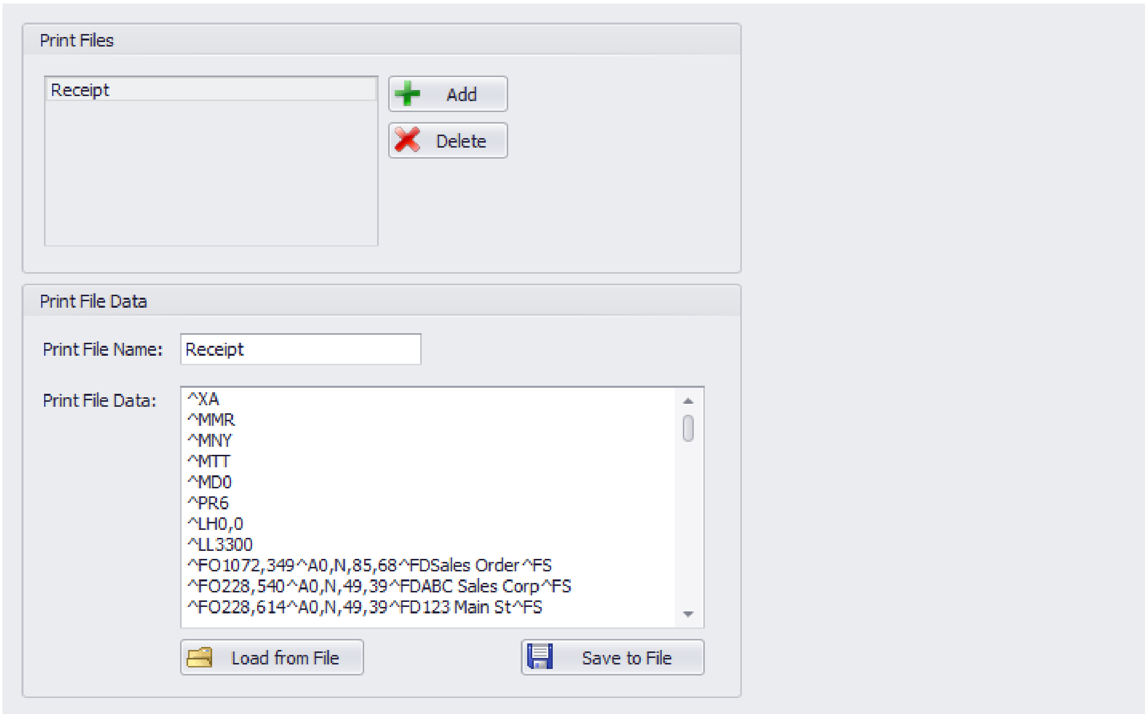
The client always starts with the Default language. If you call SetLanguage in an In-Prompt Script, the client will change to use the language you specified. Then, all elements will draw their display text from the corresponding column in the String Table. This allows you to easily switch languages at run time.

You can also export and import string table data via CSV files. This allows you to send your text out for translation and re-import the results.

In addition to the String Table entries for Elements, you can also add your own entries to the String Table. You then use the GetStringTableEntry function in an in-prompt script to retrieve the text. You could use these (for example) for the text to be displayed in a message box, or for any fixed text that is not attached to an element.

3.8 Print Files

ITScriptNet allows printing to IrDA, RF, Serial or Bluetooth printers from portable devices that can support printing (the devices must have the proper ports or radios). There are three methods to print to a printer, one for each of the Print functions described in the Print Functions section of the Function Reference of this User Guide. One of the methods utilizes Print Files which are created from the Print Files screen..



Receipt

Print File Name: Receipt

Print File Data:

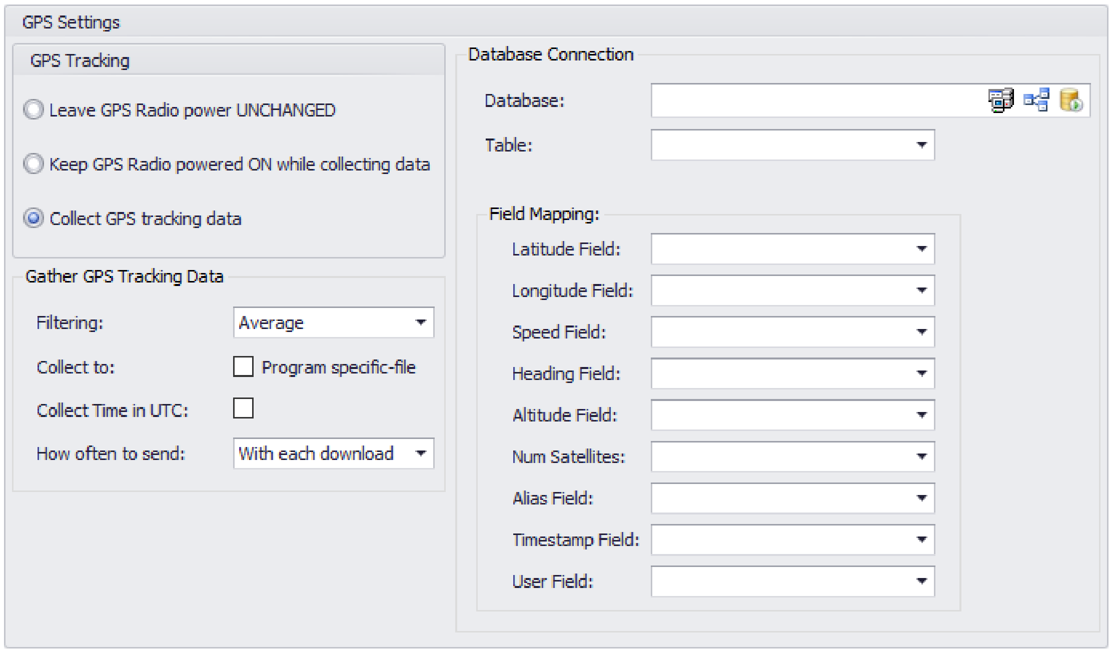
```
^XA
^MMR
^MNY
^MTT
^MD0
^PR6
^LH0,0
^LL3300
^FO1072,349^A0,N,85,68^FDSales Order^FS
^FO228,540^A0,N,49,39^FDABC Sales Corp^FS
^FO228,614^A0,N,49,39^FD123 Main St^FS
```

Any Print Files that have been configured will be displayed in the list. Print Files included in a data collection program will be uploaded to a device with the program, validation files, and other files necessary for the program to function on the device.

The Print File Data is the actual printer command to be sent to the printer. Calling the xxxPrtPrint functions will send this data to the printer. Before the data is sent, the client will perform variable substitution. This will replace any global variable names (@Varname@) with the value of that variable, and element values ($Prompt.Element$) with the value of the element. Then the data is sent to the printer.

3.9 GPS Tracking

This screen configures whether GPS data will be collected by the device.


GPS Settings

GPS Tracking

- ( ) Leave GPS Radio power UNCHANGED
- ( ) Keep GPS Radio powered ON while collecting data
- (●) Collect GPS tracking data

Gather GPS Tracking Data

Filtering:	Average
Collect to:	☐ Program specific-file
Collect Time in UTC:	☐
How often to send:	With each download

Database Connection

Database:

Table:

Field Mapping:

Latitude Field:

Longitude Field:

Speed Field:

Heading Field:

Altitude Field:

Num Satellites:

Alias Field:

Timestamp Field:

User Field:

Leave GPS Radio Power Unchanged
This option does not change the GPS radio power. If you have powered on or off the radio outside of ITScriptNet, it will remain that way. No GPS tracking data will be collected.

Keep GPS Radio Powered ON while collecting data
This option will connect to the GPS radio to power it on and leave it on as long as this program is running. When the program exits, the GPS radio will be powered off again. No tracking data is collected with this option.

Collect GPS tracking data
This option will connect to the GPS radio to power it on and leave it on as long as this program is running. When the program exits, the GPS radio will be powered off again. Tracking data will be collect as long as the program is running. The options for how GPS tracking data will be collected and processed will be enabled if this option is chosen.

Gather GPS Tracking Data
<u>Filtering</u>
This option determines how much GPS data will be collected and how accurate the tracking data will be. There are 5 choices:

Less Data. This mode collects a location a few time per minute. This is the least accurate, but collects the least data.

Average. This mode collects reasonably precise data. When the device is sitting still, a data point is collected every few minutes. When the device is moving, a location point will be collected every few seconds when there is a significant heading or speed change.

More Precise. This mode collects the most precise tracking, but collects a large amount of data. A data point will be collected as often as once per second when there are speed and heading changes, and every 15 or so seconds when moving at a steady speed and direction.

Once Per Send. This mode collects one location point for each Send interval set by the How Often To Send option.

Once Per Stop. This mode collects one location point if the device speed is below 1 mph for more than 45 seconds. Only one point is collected regardless of how long the device is stopped.

The choice of filter depends of what you are doing with the collected tracking data. If you simply want to have a rough idea of where your device was at a particular time, the Less Data option may be accurate enough. If you want to overlay your tracking data on a map, the Average or More Precise options will be more suitable. If you just want to collect general information about where the device was, the Once Per Send or Once Per Stop options will collect the least amount of data.

Collect To Program Specific File
If this option is unchecked (default), then all GPS tracking data for all data collection programs is collected together and processed at the same time. For example, if a user runs Program A for awhile and collects GPS Tracking data, then switches to Program B and collects more tracking data, all of the collected tracking data will be processed together. If this option is checked, the GPS tracking data collected for this program is kept separate from the others and is processed separately.

How often to send
This option controls how the GPS tracking data will be sent back to the PC. You can select With Each Download to process the tracking data along with the standard collected data for the program. Whether the data is downloaded over a USB connection using the Download Utility or over an OMNI Server connection, the collected GPS tracking data will be sent and processed at that time. The other option is to select a time interval to send the tracking data. If a time interval is specified, the device will attempt to send the tracking data to the OMNI Server automatically in the background. You can select time intervals from 60 seconds to 60 minutes.

Database Connection
These options are used to configure the processing of your GPS tracking data. This allows you to specify a database connection and map the location fields to fields in your database.

Database
This field specifies either an OLEDB Database connection string or an ODBC Data Source Name. You can use the Data Link Editor button to construct a connection string, or the Browse DSN button to select a data source name.

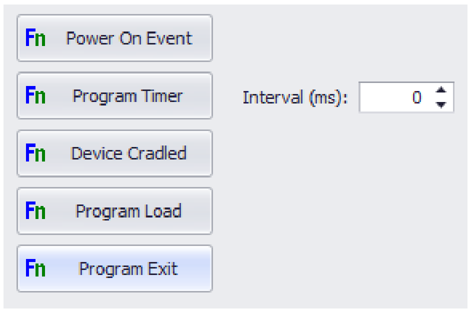
Table
Once you have set a valid connection, the Table combobox will be populated with the tables in your database. Select the table that will receive your GPS tracking data.

Field Mapping
This list is used to map the GPS location fields to the fields in your database. When you click on a field, a dropdown list will appear that contains all of the fields found in the Table you specified. You can select a field or choose the blank item if you do not want to save that particular field. For example, you might be interested only in the latitude and longitude, and ignore the number of satellites and altitude.

GPS Field Mapping
The data for the User Field is the value set using the GPSSetUserField function, otherwise it is blank.

3.10 Program Events

This screen is used to enter scripts that are run on specific program Events. These Events will run no matter which Prompt is loaded.



Power On Event
This Event occurs when the device is powered on after being suspended.

Program Timer
This is a global timer that executes periodically, using the 'Interval' specified to the right of the button. An interval of 0 disables the timer.

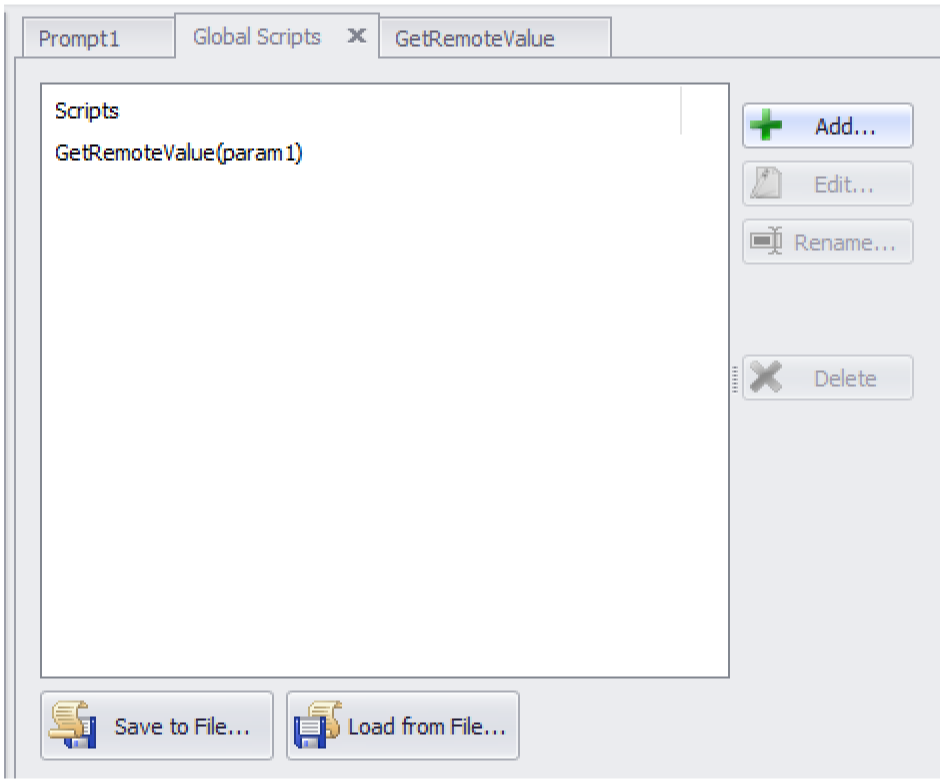
Device Cradled
This Event executes whenever the device is placed on external power.

Program Load
This Event runs when the program is started, before any Scripts on the first Prompt are loaded.

Program Exit Event
This Script runs when the program is about to exit. You can use this Event to prevent the program from exiting. By returning "0" from this Script, the program will not exit. Return non-zero to allow exiting.

3.11 Remote Scripts

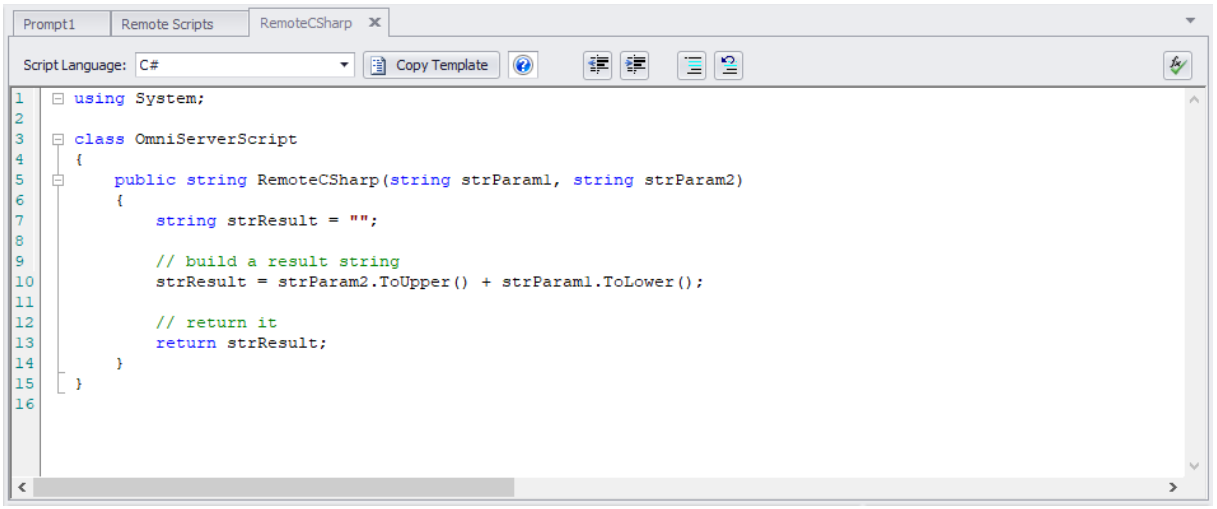
The Remote Script screen allows you to define scripts that can be called from the mobile device, but executed on the Omni Server. This can be used to get data from the server, or to perform complex calculations or lookups that cannot be done on the mobile device.



Press Add to create a new Remote Script. The parameters you enter in the script name will be passed as parameters to the script.

You can use the Edit button to edit the script itself, and the Rename button to change the name and parameters of the script.

Since Remote Scripts execute on the server, they can be written in vbscript, javascript, C# or VB.Net.



Script Language: C#

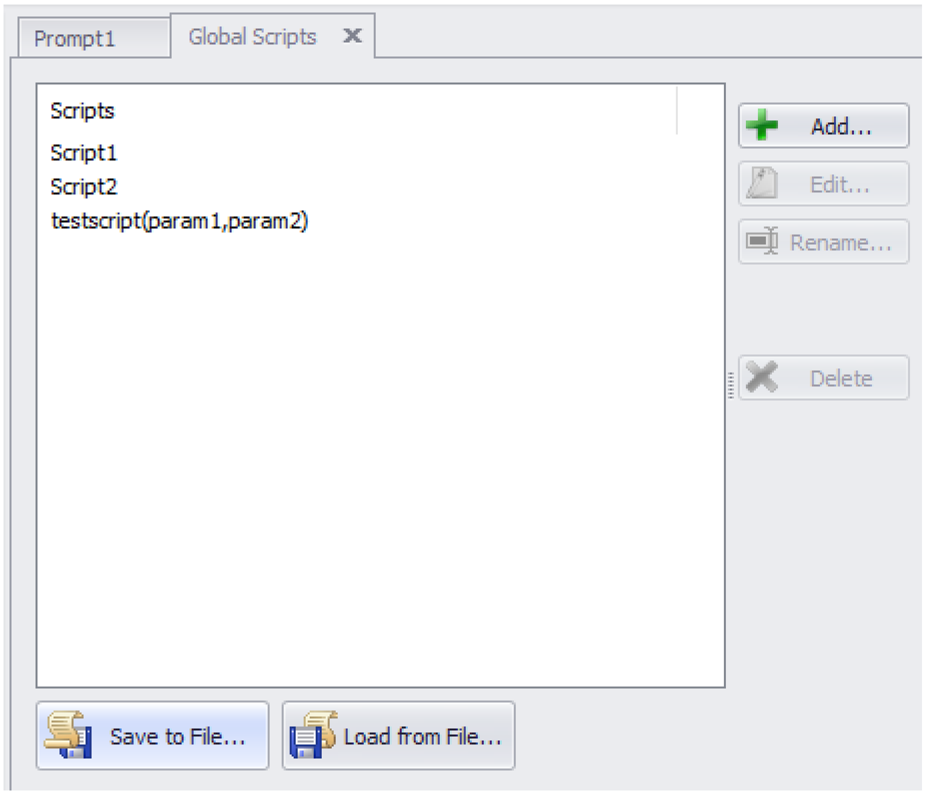
```
using System;

class OmniServerScript
{
    public string RemoteCSharp(string strParam1, string strParam2)
    {
        string strResult = "";

        // build a result string
        strResult = strParam2.ToUpper() + strParam1.ToLower();

        // return it
        return strResult;
    }
}
```

You select the language you want from the Script Language drop down. When starting a new script, use the Copy Template button to create a blank script with the correct wrapper class or structure. Then you can add your custom code to the script.

Parameters are always passed as strings.

3.12 Global Scripts

The Global Scripts screen allows you to create scripts that can be called from anywhere in the program, using the GlobalScript function. This allows you to encapsulate common functionality that is used in several places in your program.



Scripts
Script1
Script2
testscript(param1,param2)

Press Add to create a new Global Script. The parameters you enter in the script name will be available as Local Variables in the script.

You can use the Edit button to edit the script itself, and the Rename button to change the name and parameters of the script.

3.13 Reports

Reports are a feature in ITScriptNet that you can use to print labels, receipts, and more. You can visually design the reports to include such elements as text, images, and barcodes. Then you can use In-prompt script functions at runtime to generate printer data commands, and to send the data to the printer. Reports support both fixed size and variable length printing.

Report Designer
You use the Report Designer to layout what your report should look like. The report designer includes these elements:
- Text
- Images
- Barcodes
- Boxes
- Tables
- Subreports

The printed values of these elements, along with other properties, can be overriden by property scripts when the report is converted to printer commands.

Converting and Printing the Report
There are two steps you must follow to print a report. First the report must be converted to a printer command file, and second the printer command file must be sent to the printer.

There are four conversions supported:
- ZPL for Zebra compatible printers.
- PCL for HP compatible Laser and Deskjet printers.
- PNG to convert the report into an image.
- HTML to convert to a web page that could be emailed or viewed on screen.

Each conversion function has two forms - one to convert a report designed and contained within the ITBX, and one to convert a report stored in an external file. This second form can be used if you have a common report that is shared between more than one ITBX.

Once the report has been converted, you can send the printer command file to the printer using the Serial, Bluetooth, or RF printing functions. For example, BtPrtFile() will send a file to a Bluetooth printer.

Part IV

4 Other Notes

The following topics contain some notes about specific features.

4.1 Clicking on Shapes

Shapes can only be clicked if they are not transparent (Opaque or gradient). If the shape is transparent, al clicks on the shape will be ignored. This means that a control underneath a transparent shape can be clicked. A control underneath an opaque shape cannot be seen or clicked.

4.2 Reports

Reports can be exported to a variety of formats, each with its own unique conversion characteristics. Be realistic about your expectations for a printed report. For example, do not expect a thermal transfer printer to print gray scales or fine detail, due to the limitations of the printing technology.

Report Background Colors
Background colors are supported on HTML and PNG formats only. On the other conversions, the background color is ignored.

Text Fields
Fonts: Since the conversion can take place on a mobile device that is not necessarily equipped with all of the same fonts as your PC, there is a limited selection of fonts available. These fonts are common to most of the conversion formats.

You can select Helvetia (a proportionally spaced, san-serif typeface), Times Roman (a proportionally spaced, serif typeface), and Courier (a monospaced typeface).

The final output can vary from one conversion method to another. You may have to make adjustments to your design to get the best output for your application.

- HTML
  - Helvetica maps to Arial with fallback to ‘sans-serif’.
  - Times Roman: maps to Times Roman with fallback to ‘serif’.
  - Courier: maps to Courier New with fallback to ‘monospaced’.

- PNG
  - Helvetica
    - PC: maps to Arial.
    - Pocket PC / Windows CE: maps to Tahoma which is the only sans-serif font available.
    - Android: maps to the default font

  - Times Roman
    - PC: maps to Times Roman.
    - Pocket PC / Windows CE: maps to ??
    - Android: maps to the default font

  - Courier
    - PC: maps to Courier New.
    - Pocket PC / Windows CE: maps to ??
    - Android: maps to the default font

- PCL
  - Helvetica: Maps to the built-in Arial font.
  - Times Roman: Maps to the built-in CG Times font.
  - Courier: Maps to the built-in Lineprinter font.

- ZPL
  - Both Helvetica and Times Roman map to the built-in scalable font (^A0). There is no built-in Serif font in ZPL.
  - Courier maps to the monospaced Bitmapped fonts. The conversion process selects the bitmapped font which best matches the requested font size with the smallest multiplication factor.

Colors: Not all conversion types support color.

- HTML: Text foreground, background, and border colors are fully supported.
- PNG: Text foreground, background, and border colors are fully supported.
- PCL: All PCL conversions assume a black-and-white printer, not color. Text and the Borders are always black. The background color will be converted to grayscale, based on the background color selected.
- ZPL: Thermal printers do not support grayscale or color. However, if the text color is a lighter intensity than the background, then the text will be printed in reverse (white on black). The border is always black.

Justification and Text Wrapping: Each conversion type has a different set of fonts. As a result, it is not necessarily possible to get the exact text sizes needed to wrap or right-justify text. For example, the PCL Lineprinter font characters are wider than the PC's Courier characters. The conversion makes an attempt to calculate the text sizes, but final results may not match the designer layout. You may have to make adjustments to your design to get the final result to look just like you want. HMTL and PNG outputs follow more closely to the designer than to the PCL or ZPL outputs.

4.3 Validation File Query Parameters

ITScriptNet V3.x supported Bracket { } parameters in validation file queries. These were directly substituted with the supplied value when the validation file was generated, either by prompting the user on the Validation Files screen, or with the Match Parameters when CreateValidationRemote was called. These parameters are still supported in Indago for backwards compatibility, but there are other parameter types supported as well.

Bracket Parameter problems
The new Query Designer in Indago requires that the query be a valid SQL Query for the database type that has been selected (ODBC, OLEDB, or MSSQL). The Bracket characters are not valid characters by themselves in any of these. As long as the parameter name is wrapped in single quotes, the Query Parser can treat it as a string and everything works fine. Example:

```
WHERE Field = ‘{Filter}’
```

However, this limits the places where the bracket parameters can be used. For example, you cannot readily use them in an IN clause:

```
WHERE Field IN ({Filter})
```

is invalid syntax.

```
WHERE Field IN (‘{Field}’)
```

is valid, but then you have to remember to format the values for the {Field} parameter carefully to take the single quotes into account. For example, instead if calling

```
CreateValidationRemote(“file”, “Field”, “’ABC’, ‘DEF’”) <- each value wrapped in single quotes
```

you would have to use

```
CreateValidationRemote(“file”, “Field”, “ABC’, ‘DEF”) <- first and last single quotes skipped.
```

New Parameter Types
There are other character types that the Query Builder will accept as valid. These include:

:PARAM for OLEDB and ODBC.

The Query Builder will accept parameters starting with : without quoting. Note that there is only a leading ‘:’ character, and not one at the end. The parameter ends at the first character that is not a letter, number, or underscore.
This is still a direct text substitution parameter, exactly like Bracket parameters. The only difference is that this is valid SQL Syntax for the ODBC and OLEDB drivers. For example:

```
WHERE Field = :Param
```

Or

```
WHERE Field IN (:Param)
```

You are responsible for supplying the single quotes for text fields when calling CreateValidationRemote, however.

```
CreateValidationRemote(“file”, “:Param”, “’ABC’, ‘DEF’”) <- each value wrapped in single quotes
```

This parameter type is not supported for direct MSSQL connections, just ODBC and OLEDB.

? Positional Parameters
New in V4.x is the ability to specify actual query parameters. These are not text substitutions, but the query command is parameterized and values specified. This is a more robust method of specifying values to the query and prevents the possibility of SQL Injection.
ODBC and OLEDB only support positional parameters, not named parameters. This means that the parameters are specified by the ‘?’ character, and the value are matched to the parameter by position. For example:

```
WHERE Field1 = ? OR Field2 = ?
```

has two parameters. The corresponding CreateValidationRemote call would be:

```
CreateValidationRemote(“file”, “?”, “ABC”, “?”, “DEF”)
```

Note that because these are parameters instead of text substitutions, there is no need to wrap the values or parameters in single quotes.
When the validation file is generated in the Program Designer, the parameter value prompt will list the parameters as ?1, ?2, ?3, etc so you can better distinguish them. However, in the query text you must simply use ‘?’. For CreateValidationRemote, you can specify the parameters as ‘?’ or ‘?1’, ‘?2’, etc. Both forms are acceptable.

@ Named Parameters
MS SQL Server supports named parameters. The Query Builder only recognizes them if you make a MSSQL connection. Named parameters will not work if you use an ODBC or OLEDB connection to SQL Server.

Named parameters are can be in any order and repeated multiple time. For example:

```
WHERE Field1 = @Param OR Field2 = @Param
```

To generate the file, call:

```
CreateValidationRemote(“file”, “@Param”, “ABC”)
```

You can specify any number of named parameters.

Summary
Here is a summary of which types of parameters are supported for each connection type:

	ODBC	OLDB	MSSQL
{Bracket}	Yes	Yes	Yes
:Param	Yes	Yes	No
?	Yes	Yes	No
@Named	No	No	Yes

Part V

5 Device Specific Notes

Not all features are available on all hardware platforms. The following sections contain notes about how some features behave on specific hardware.

5.1 Fonts

Using Fonts on Android

Android devices earlier than V4 only have 3 fonts installed by default:
- Droid Serif
- Droid Sans Serif
- Droid Monospaced

In V4, the Roboto font was added and is the default font.

Since the fonts on the PC are not the same as the fonts on Android, the Android font mapper will select a font that best matches the requested styles. This means you will usually get Droid Sans Serif on older devices or Roboto on newer devices. If you want to use other fonts, you must add the TTF files as support files so they are loaded with the program. The ITScriptNet client will then be able to load and use the fonts.

Font Name Translations
You can download the Droid fonts and install them on your Designer PC. However, fonts are selected on Android using the Family Name, not the font file name. For example, the Droid Sans font is selected using “sans serif” as the family name. This requires a translation of the names from the Font Name to the Family Name. The following font names will be translated on the Android device as follows:

Droid Serif: “serif”
Droid Sans: “sans serif”
Droid Sans Mono: “monospaced”

Any other font name which does not correspond to a TTF file will be mapped to the default font.

Roboto Font Files
The Roboto font files are free to download and can be installed on the Designer PC. There are several variations of the font that are installed. This can cause some problems because several of the font files share the same Family Name. For example, ROBOTO-LIGHT.TTF and ROBOTO-LIGHTITALIC.TTF both have the Family Name ‘Roboto Lt’. If you deploy all of the Roboto font files to the device, the font mapper cannot distinguish between different files that share the same font family. This can cause the wrong font to be displayed, resulting in Italic or Bold text being displayed where Regular text was called for, or vice versa. To prevent this, only deploy these Roboto font files:

Font File	Family Name	Comments
ROBOTO-REGULAR.TTF	Roboto	Covers Bold
ROBOTO-BLACK_0.TTF	Roboto Bk	
ROBOTO-CONDENSED.TTF	Roboto Cn	
ROBOTO-LIGHT.TTF	Roboto Lt	Covers Medium
ROBOTO-THIN_0.TTF	Roboto Th	

For the Bold or Italic variants, use the Bold or Italic settings rather than deploying the Bold or Italic font files.

Using Fonts on Windows CE/ Windows Mobile

The only fonts that are guaranteed to be installed on Windows CE/Windows Mobile devices are Tahoma and System.

Additional fonts can be used if the TrueType Font file (.TTF) is attached to the program as a support file and loaded to the device. If the TTF file is not sent to the device, Windows will make a best match of the font based on the styles you selected, but the result may not match what was expected. If you use a font other than Tahoma or System, be sure to attach the TTF file to the program as a support file.

Using Fonts on the PC

The font must be installed in Windows on the Designer PC before you can select it in the Font Selection drop down boxes.

The font does not need to be installed in Windows on a PC running the PC Client, however. You can simply deploy the TTF file along with the program. Attaching the .TTF file as a Support File will take care of this.

5.2 Screen Rotations

There is a Screen Rotation setting on the Prompt Settings screen. The possible settings are:

- Default: The device's sensor determines the rotation if the device supports automatic screen rotation. Otherwise no change is made to the screen rotation.
- Portrait: The device is set into Portrait orientation, and the screen is locked into this orientation.
- Landscape: The device is set into Landscape orientation, and the screen is locked into this orientation.

This setting applies to Windows Mobile, Windows CE, and Android. It has no effect on the Simulator or PC Client.

5.3 Subprompt Scrolling

All platforms support scrolling subprompts either Vertically or Horizontally.

Windows CE and Windows Mobile only support scrolling in both directions at the same time. If you select Vertical or Horizontal scrolling, the subprompt will still scroll in both directions if the subprompt is large enough to require scrolling.

Android only supports scrolling in one direction. If you select Both Scroll on Android, it will only scroll vertically.

Desktop Windows supports all scrolling options.

5.4 Email

There are a few platform-specific limitations on sending emails.

Windows CE / Windows Mobile

- You cannot attach files to emails in Windows CE/Windows Mobile.
- You must specify the Domain even if a Username and Password are not needed.

5.5 Keyboard

Android

Android Soft Keyboards do not trigger the TextChanged event like Hardware keyboards do. This means that the TextChanged event will not fire reliably when using the Soft Keyboard.

This also means that Limit Keys may not always prevent a user from typing a character that should not be allowed. For example, with a textbox set to Uppercase Only, the user may be able to type lowercase letters by switching the soft keyboard into lowercase mode. We recommend that you verify or convert the text in a textbox to make sure that it does not contain an characters that you don’t want. You can do this in LostFocus or in the Validation Event.

GetKeyboardMode is not supported on Android and always returns 3 (Lowercase Alpha).

5.6 Radio Modes

Android
Applications in Android cannot enable or disable the Phone (unless the phone is rooted). Therefore, RadioSetMode can only enable and disable the WiFi and Bluetooth radios. The Phone will remain on at all times. The user can put the phone in Airplane Mode through the Settings applet, but that is the only supported way to turn off all radios.

Windows Mobile
RadioSetMode works on Windows Mobile, but the RadioGetMode state doesn't always refresh afterward.

Windows CE
RadioGetMode and RadioSetMode are not supported on Windows CE because there is no standard interface for turning the radios on or off. This is a manufacturer specific operation on Windows CE.

PC Client/Simulator
RadioGetMode and RadioSetMode are not supported on the PC.

5.7 Flash LEDs

Android
Android does not allow applications to directly control the LEDs. Instead, FlashLEDs vibrates the phone briefly.

Windows Mobile/Windows CE
FlashLEDs turns on the LEDS briefly. If the device is equipped with a Vibrator, it will also be operated briefly.

PC
FlashLEDs is not supported on the PC.

5.8 Powerdown Mode

Android

Android does not support Unattended mode.

5.9 RAS Support

In V3, we supported RAS to dial a cellular data connection on Windows Mobile or CE. However in Indago, this has changed significantly.

Android
Android has no concept of dialing a data connection. It just uses cellular data when necessary. Therefore, the RAS functions don't do anything on Android.

Window Mobile
Since the new OMNI Communications method is more akin to a Web Service than the old V3 sockets method was, Windows Mobile Connection Mananger will automatically establish a cellular data connection if it feels it needs to. Therefore, the RAS functions aren't really needed on Windows Mobile. However, we did implement them in case there is a circumstance where a user needs to establish a connection manually.

Windows CE
RAS is supported on Windows CE, although GPS is not.

PC
RAS is not supported on the PC.

So the final result is that the RAS functions are implemented for Windows Mobile and Windows CE, but will probably not be needed anymore.
Also, we removed the Data Connection options from the GPS Tracking screen since they don't have any effect on any device anymore.

Part VI

6 Function Reference

6.1 Conversion Functions

Asc	
Syntax:	Asc(<Char>)
Parameters:	
<Char>	A string containing the character to be converted
Returns:	Returns the ASCII character code for the first character in <Char>
Examples:	Asc("A”) = 65
Notes:	Converts a character to its ASCII equivalent

Chr	
Syntax:	Chr(<Num>)
Parameters:	
<Num>	Numeric value to be converted
Returns:	Returns the character associated with a specified ASCII character code
Examples:	Chr(65) = "A"
Notes:	The value of <Num> should be between 0 and 255

Format	
Syntax:	Format(<Num> , <Numdecimals>, <Sep>, [thousep], [decsep])
Parameters:	
<Num>	The number to format
<Numdecimals>	How many decimal places to keep
<Sep>	Whether to use a thousands separator
[thousep]	Optional character to use as thousands separator
[decsep]	Optional character to use as decimal separator
Returns:	Returns the number as a string with the number of decimals specified
Examples:	Format(@Num@, 2, 1) = "17,345.21" where @Num@ = 17345.2142
Notes:	Set the <Sep> parameter to 1 to include ',' as the thousands separator. If <Sep> is set to 0, no separator will be used. Optionally, you can specify the Thousands separator and Decimal Separator. The default Thousands separator is ',' and the default Decimal separator is '.' if not specified.

FormatNumberCustom	
Syntax:	Format(<Num> , <FormatString>)
Parameters:	
<Num>	The number to format
<FormatString>	A .Net formatting string
Returns:	Returns the number as a string formatted with the .Net formatting string
Examples:	FormatNumberCustom(@MyNum@, "N") Returns "17,345.2142" if @MyNum@ = 17345.2142
Notes:	Valid formatting strings can be any of the .Net Standard or Custom Numeric formatting strings.

HexDecode	
Syntax:	HexDecode(<String>)
Parameters:	
<String>	The hex encoded string to decode.
Returns:	Returns the the number represented by the hex encoded <String>.
Examples:	HexDecode("007F") returns 127
Notes:	If the string contains invalid digits, this function returns 0

HexEncode	
Syntax:	HexEncode(<String>, <Digits>)
Parameters:	
<String>	The string to be converted.
<Digits>	The number of digits to pad.
Returns:	Returns the the hex representation of the number, padded with zeros to the specified number of digits.
Examples:	HexEncode("127", 4) returns 007F
Notes:	If the string contains invalid characters, returns 0

SHA2	
Syntax:	SHA2(<String>, <Digest>)
Parameters:	
<String>	The string to be converted.
<Digits>	The digenst to use. Valid values are 256, 384, 512.
Returns:	Returns the the hex representation of the SHA2 Hash of the input string..
Examples:	SHA2("127", 256) returns the SHA2 hash of the input
Notes:	If the digest is not one of 256, 384, or 512, then a digest of 256 will be used.

Val	
Syntax:	Val(<String>)
Parameters:	
<String>	The string to be converted
Returns:	Returns an integer representation of the string in <String>
Examples:	Val("124a6") returns 124
Notes:	Converts the string from left to right, stopping at the first non-numeric character.

6.2 Date/Time Functions

BuildDate	
Syntax:	BuildDate(<Year>, <Month>, <Day>, <Hour>, <Minute>, <Second>)
Parameters:	
<Year>	The year to use for building the date / time.
<Month>	The month to use for building the date / time.
<Day>	The day to use for building the date / time.
<Hour>	The hour to use for building the date / time.
<Minute>	The minute to use for building the date / time.
<Second>	The second to use for building the date / time.
Returns:	The resulting date time.
Examples:	BuildDate (2005, 1, 9, 11, 13, 52) returns "01/09/2005 11:13:52".
Notes:	Calculates the date / time resulting from the parameters specified.

Date	
Syntax:	Date([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Date() returns “08/18/2005” on August 18th, 2005.
Notes:	Returns the current date if the parameter is not specified.

DateAdd	
Syntax:	DateAdd(<Datetime>, <Interval>, <Amount>)
Parameters:	
<Datetime>	Starting date / time.
<Interval>	The type of interval to add. Valid options are "Y" (years), "M" (months), "D" (days), "H" (hours), "N" (minutes), "S" (seconds).
<Amount>	The number to add to the starting date / time.
Returns:	The resulting date time.
Examples:	DateAdd("01/05/2005 11:13:52", "D", 4) returns "01/09/2005 11:13:52".
Notes:	Calculates the date / time resulting from adding the <Interval> to the <Datetime>. Note that <Datetime> must be in the format MM/DD/YYYY HH:NN:SS.

DateCompare	
Syntax:	DateCompare(<Datetime1>, <Datetime2>)
Parameters:	
<Datetime1>	The starting date for the comparison.
<Datetime2>	The ending date for the comparison.
Returns:	A number indicating the result of the comparison.
Examples:	DateCompare("01/10/2005 11:13:52", "01/09/2005 06:32:51") returns 1.
Notes:	Compares two dates. Returns -1 if <Datetime1> is earlier than <Datetime2>. Returns 1 if <Datetime1> is later than <Datetime2>. Returns 0 if both dates are equal. Note that <Datetime> must be in the format MM/DD/YYYY HH:NN:SS.

DateDiff	
Syntax:	DateDiff(<Datetime1>, <Datetime2>, <Interval>)
Parameters:	
<Datetime1>	Starting date / time
<Datetime2>	Ending date / time
<Interval>	Type of interval to calculate. Valid options are "Y" (years), "M" (months), "D" (days), "H" (hours), "N" (minutes), "S" (seconds).
Returns:	The time interval between the two dates.
Examples:	DateDiff("01/05/2005 11:13:52", "01/05/2005 12:15:33", "H") returns 1.
Notes:	Calculates the time interval from <Datetime1> to <Datetime2>. Note that <Datetime> must be in the format MM/DD/YYYY HH:NN:SS. To get a positive number for the result, <Datetime1> should be less than <Datetime2>.

DateFormat	
Syntax:	DateFormat(<Datetime>, <Format>)
Parameters:	
<Datetime>	The date / time to format.
<Format>	Specifies the format to return.
Returns:	A date / time formatted according to the specification.
Examples:	DateFormat("01/10/2005 11:13:52", "M") returns “01/10/2005”.
Notes:	Reformats a Date/Time for display purposes. Valid format specifiers are: M: MM/DD/YYYY S: MM/DD/YY D: DD/MM/YYYY E: DD/MM/YY T: HH:NN:SS (24-hour) A: HH:NN:SS am/pm H: HH:NN (24-hour) P: HH:NN am/pm X: YYYYMMDDHHNNSS Q: YYYY-MM-DD HH:NN:SS This function is for display only. Note that <Datetime> must be in the format MM/DD/YYYY HH:NN:SS.

Day	
Syntax:	Day([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Returns:	Returns the day of the month
Examples:	Day() returns 18 on the 18th of the month.
Notes:	Returns the current day if the parameter is not specified.

DayOfWeek	
Syntax:	DayOfWeek([Datetime])
Parameters:	
[Datetime]	Optional. If specified, this is the datetime to use for calculating the Date of the Week.
Returns:	The day of the week for the specified datetime or current date.
Examples:	DayOfWeek ("01/09/2005 11:13:52") returns 1.
Notes:	Returns the Day of the Week, as a number, for a date/time with Sunday = 1, Monday = 2, etc. If the [Datetime] parameter is specified, it will be parsed for the date. If it is not specified, the current date will be used. Note that [Datetime] must be in the format MM/DD/YYYY HH:NN:SS.

DayOfWeekName	
Syntax:	DayOfWeek([format], [Datetime])
Parameters:	
[Format]	If [format] is 1, the full month name is returned. If [format] is 2, an abbreviated name is returned.
[Datetime]	Optional. If specified, this is the datetime to use for calculating the Date of the Week.
Returns:	The day of the week for the specified datetime or current date.
Examples:	DayOfWeek (1, "01/09/2005 11:13:52") returns "Sunday".
Notes:	Returns the Day of the Week, as a number, for a date/time with Sunday = 1, Monday = 2, etc. If the [Datetime] parameter is specified, it will be parsed for the date. If it is not specified, the current date will be used. Note that [Datetime] must be in the format MM/DD/YYYY HH:NN:SS.

DayOfYear	
Syntax:	DayOfYear([Datetime])
Parameters:	
[Datetime]	Optional. If specified, this is the datetime to use for calculating the Date of the Year.
Returns:	The day of the year (Julian date) for the specified datetime or current date.
Examples:	DayOfYear ("01/09/2005 11:13:52") returns 9.
Notes:	Returns the Day of the Year, as a number, for a date/time with Jan 1st = 1. If the [Datetime] parameter is specified, it will be parsed for the date. If it is not specified, the current date will be used. Note that [Datetime] must be in the format MM/DD/YYYY HH:NN:SS.

GetTickCount	
Syntax:	GetTickCount( )
Parameters:	None
Examples:	@ret@ = GetTickCount( )
Notes:	Returns the number of milliseconds since the device was booted. Does not include time that the device was suspended. The solution of this function depends on the device, but is generally 18ms. This function wraps around every 49.7 days.

Hour	
Syntax:	Hour([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Hour returns 14 from 2:00pm to 2:59pm.
Notes:	The result is always in 24-hour time. Returns the current hour if the parameter is not specified.

Minute	
Syntax:	Minute([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Minute returns 15 at quarter after each hour.
Notes:	Returns the current minute if the parameter is not specified.

Month	
Syntax:	Month([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Month() returns 8 in August.
Notes:	Returns the current month if the parameter is not specified.

MonthName	
Syntax:	MonthName([format], [datetime])
Parameters:	
[format]	If [format] is 1, the full month name is returned. If [format] is 2, an abbreviated name is returned.
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	MonthName(2) returns "Aug" in August.
Notes:	Returns the current month if the parameter is not specified.

Now	
Syntax:	Now()
Parameters:	None
Returns:	The current date and time, in the format MM/DD/YYYY HH:MM:SS
Examples:	Returns “08/18/2005 23:13:26” at 11:13:26pm on August 18th, 2005.
Notes:	Returns the current time and date.

Second	
Syntax:	Second([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Second() returns 30 at 30 seconds past each minute.
Notes:	Returns the current second if the parameter is not specified.

SetClock	
Syntax:	SetClock(<datetime>)
Parameters:	
<datetime>	A DateTime string. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	SetClock("03/18/2008 11:15:25")
Notes:	Sets the device clock using the current time zone.

Time	
Syntax:	Time([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Time() returns "23:13:26" at 11:13:26pm.
Notes:	Returns the current time if the parameter is not specified.

UTCNow	
Syntax:	UTCNow( )
Parameters:	None
Returns:	Returns the current date and time in UTC, in the format: MM/DD/YYYY HH:NN:SS
Examples:	UTCNow( )

Notes:	Converting between UTC and Local Time:
	localtime = UTC + tzOffsetMinutes
	UTC = localtime - szOffsetMinutes

Year	
Syntax:	Year([datetime])
Parameters:	
[datetime]	An optional DateTime string. If not specified, the current system date/time is used. Must be in the format MM/DD/YYYY HH:MM:SS.
Examples:	Year() returns 2005 in 2005.
Notes:	Returns the current year if the parameter is not specified.

6.3 Element Functions

AddItem	
Syntax:	AddItem (<element>, <text>, <data>)
Parameters:	
<element>	The name of the element to check.
<text>	The text to insert into the list.
<data>	The Item Data to use for the new list item.
Returns:	Nothing
Examples:	AddItem ("Prompt1.ComboBox", "New Item", "123")
Notes:	For Combo Boxes and List Boxes, adds the item with <text> and <data> to the end of the list.

BarcodeScanCamera	
Syntax:	BarcodeScanCamera(<Elementname>)
Parameters:	
<Elementname>	The name of the element to receive the scan.
Returns:	Nothing
Examples:	BarcodeScanCamera("prompt.element")
Notes:	Starts the camera-based barcode scanner and puts the result in <elementname>. The <elementname> must specify an element that supports scanning and has the Input Source and Barcode Scanner permit scanning, or the function does nothing.

Clear	
Syntax:	Clear(<Elementname>)
Parameters:	
<Elementname>	The name of the element to clear.
Returns:	Nothing
Examples:	Clear("prompt.element")
Notes:	Clears the multiprompt element. Elements are referenced using the syntax “prompt.element”.

DeleteItem	
Syntax:	DeleteItem (<element>, <index>)
Parameters:	
<element>	The name of the element to check.
<index>	The index of the item to delete.
Returns:	Nothing
Examples:	DeleteItem ("Prompt1.ComboBox", 2)
Notes:	For Combo Boxes, List Boxes and Grids, deletes the item at position <index> from the list.

Disable	
Syntax:	Disable(<Elementname>)
Parameters:	
<Elementname>	The name of the element to disable.
Returns:	Nothing
Examples:	Disable("prompt.element")
Notes:	Disables a multiprompt element that had been enabled. Elements are referenced using the syntax “prompt.element”.

Enable	
Syntax:	Enable(<Elementname>)
Parameters:	
<Elementname>	The name of the element to enable.
Returns:	Nothing
Examples:	Enable("prompt.element")
Notes:	Enables a multiprompt element that had been disabled. Elements are referenced using the syntax “prompt.element”.

FindIndexByData	
Syntax:	FindIndexByData(<Elementname>, <data>)
Parameters:	
<Elementname>	The name of the element to enable.
<data>	The data to find in the list.
Returns:	The index of the first matching item.
Examples:	FindIndexByData("prompt.element", “1234”)
Notes:	For Combo Boxes and List Boxes, returns the index of the item which has the item data matching <data>. The search is case-insensitive. The first row is number 1, not 0.

FindIndexByText	
Syntax:	FindIndexByText(<Elementname>, <text>)
Parameters:	
<Elementname>	The name of the element to enable.
<text>	The text to find in the list.
Returns:	The index of the first matching item.
Examples:	FindIndexByText("prompt.element", “ABCD”)
Notes:	For Combo Boxes and List Boxes, returns the index of the item which has the text matching <text>. The search is case-insensitive. The first row is number 1, not 0.

GetCount	
Syntax:	GetCount (<element>)
Parameters:	
<element>	The name of the element to check.
Returns:	The number of items in the list.
Examples:	GetCount ("prompt.element")
Notes:	For Combo Boxes, List Boxes and Grids, returns the number of items in the list.

GetIndex	
Syntax:	GetIndex(<element>)
Parameters:	
<element>	The name of the element whose index should be retrieved.
Returns:	The index of the currently selected item
Examples:	GetIndex ("prompt.element")
Notes:	For Combo Boxes, List Boxes, Grids, and Multi Lists, returns the index of the currently selected row. The first row is number 1, not 0. For Text Boxes, returns the position of the cursor in the Text Box. The left of the first character position is 0.

GetItemData	
Syntax:	GetItemData (<Elementname>, <index>)
Parameters:	
<Elementname>	The name of the element to search.
<index>	The index of the item to retrieve.
Returns:	The item data of the matching item.
Examples:	@text@ = GetITemData ("prompt.element", 2)
Notes:	For Combo Boxes and List Boxes, retrieves the data of the item specified by <index>. The first row is number 1, not 0.

GetItemText	
Syntax:	GetItemText(<Elementname>, <text>)
Parameters:	
<Elementname>	The name of the element to search.
<text>	The text to find in the list.
Returns:	The text of the first matching item.
Examples:	@text@ = GetItemText ("prompt.element", “ABCD”)
Notes:	For Combo Boxes and List Boxes, retrieves the text of the item specified by <index>. The first row is number 1, not 0.

Hide	
Syntax:	Hide(<Elementname>)
Parameters:	
<Elementname>	The name of the element to hide.
Returns:	Nothing
Examples:	Hide("prompt.element")
Notes:	Hides a multiprompt element that had been shown. Elements are referenced using the syntax “prompt.element”.

InsertItem	
Syntax:	InsertItem (<element>, <index>, <text>, <data>)
Parameters:	
<element>	The name of the element to check.
<index>	The index of the item to insert.
<text>	The text to insert into the list.
<data>	The Item Data to use for the new list item.
Returns:	Nothing
Examples:	InsertItem ("Prompt1.ComboBox", "2", "New Item", "123")
Notes:	For Combo Boxes and List Boxes, inserts the item with <text> and <data> at the position <index>. Position 0 is before the first item, position 1 is after the first item.

IsEnabled	
Syntax:	IsEnabled(<Elementname>)
Parameters:	
<Elementname>	The name of the element to check.
Returns:	1 if the element is enabled, or 0 if not enabled.
Examples:	@ret@ = IsEnabled("prompt.element")
Notes:	Checks to see if the multiprompt element is enabled or disabled. Elements are referenced using the syntax “prompt.element”.

IsVisible	
Syntax:	IsVisible(<Elementname>)
Parameters:	
<Elementname>	The name of the element to check.
Returns:	1 if the element is visible, or 0 if not visible.
Examples:	@ret@ = IsVisible("prompt.element")
Notes:	Checks to see if the multiprompt element is visible or hidden. Elements are referenced using the syntax “prompt.element”.

Keypress	
Syntax:	Keypress( )
Parameters:	None
Returns:	Returns the character code of the key pressed to trigger the event.
Examples:	@Key@ = Keypress( )
Notes:	Valid only during the OnKeyPress Event. Calling this function during any other script or event is undefined.

Refresh	
Syntax:	Refresh(<Elementname>)
Parameters:	
<Elementname>	The name of the element to refresh.
Returns:	Nothing
Examples:	Refresh("prompt.element")
Notes:	Causes a multiprompt element to be refreshed. This causes initialization scripts to be re-run and redisplays the element. Elements are referenced using the syntax “prompt.element”.

RGB	
Syntax:	RGB(<Red>, <Green>, <Blue>)
Parameters:	
<Red>	The decimal value of the Red component of the color.
<Green>	The decimal value of the Green component of the color.
<Blue>	The decimal value of the Blue component of the color.
Returns:	A numeric value indicating the complete RGB value.
Examples:	RGB(255, 0, 0)
Notes:	Creates an RGB value from the component colors. This can be used in any of the custom color scripts.

Select	
Syntax:	Select(<Elementname>, <Index>)
Parameters:	
<Elementname>	The name of the element to select.
<Index>	Controls the selection for each element type. For Text Input boxes, an <Index> of 0 removes the selection, while an <Index> of 1 selects all text. For Comboboxes, Grids, and Listboxes, the <Index> specifies the number of the row to select. The first row number is 1, not 0.
Returns:	Nothing
Examples:	Select("prompt.element")
Notes:	Selects an item in a list, or selects the text in a Text Input box. Elements are referenced using the syntax “prompt.element”.

SetFocus	
Syntax:	SetFocus(<Elementname>)
Parameters:	
<Elementname>	The name of the element to receive the keyboard focus.
Returns:	Nothing
Examples:	SetFocus("prompt.element")
Notes:	Sets the keyboard focus to the multiprompt element. That element will now receive keyboard input. Elements are referenced using the syntax “prompt.element”.

SetGridCellColor	
Syntax:	SetGridCellColor(<grid>, <row>, <column>, <textcolor>, <backgroundcolor>)
Parameters:	
<grid>	The name of the grid element.
<row>	The row number to set the color for.
<column>	The column name in that row to change.
<textcolor>	The color of the foreground text.
<backgroundcolor>	The background color of the cell.
Returns:	Nothing
Examples:	SetGridCellColor("Prompt.grid", 2, "col2", RGB(255, 0, 0,), RGB(255, 255, 255))
Notes:	Changes the color of a particular cell in a grid. Refreshing the grid clears the colors back to the default.

SetGridRowBackgroundColor	
Syntax:	SetGridRowBackgroundColor(<grid>, <row>, <backgroundcolor>, [backgroundselectedcolor])
Parameters:	
<grid>	The name of the grid element.
<row>	The row number to set the color for.
<backgroundcolor>	The background color of the row.
[backgroundselectedcolor]	Optional. The background color for the row when selected.
Returns:	Nothing
Examples:	SetGridRowBackgroundColor("Prompt.grid", 2, RGB(255, 0, 0,), RGB(0, 255, 0))
Notes:	Set the background color of a row in a Listview/GridView/Grid. The name of the element is in <grid>. Specify the index of the row in <row>. The color to use for the background of the row goes in <backgroundcolor>. Optionally, you can specify the color to use for the row background when the row is selected in [backgroundselectedcolor]. If the selected background color is not specified, the row will use the default selected background color when selected.

SetIndex	
Syntax:	SetIndex(<element>, <index>)
Parameters:	
<Elementname>	The name of the element whose index should be set.
<index>	The index that should be selected.
Returns:	Nothing
Examples:	SetIndex ("prompt.element", 2)
Notes:	For Combo Boxes, List Boxes, Grids, and Multi Lists, sets the currently selected row to the index specified. The first row is number 1, not 0. For Text Boxes, sets the position of the cursor in the Text Box. The first character position is 0. Setting the cursor position removes any selection.

SetItemData	
Syntax:	SetItemData(<element>, <index>, <data>)
Parameters:	
<Elementname>	The name of the element whose index should be set.
<index>	The index that should be selected.
<data>	The item data that should be placed into the list.
Returns:	Nothing
Examples:	SetItemData ("prompt.element", 2, “1234”)
Notes:	For Combo Boxes and List Boxes, sets the data of the item specified by <index>. The first row is number 1, not 0.

SetItemText	
Syntax:	SetItemText(<element>, <index>, <text>)
Parameters:	
<Elementname>	The name of the element whose index should be set.
<index>	The index that should be selected.
<text>	The text that should be placed into the list.
Returns:	Nothing
Examples:	SetItemText ("prompt.element", 2, “ABCD”)
Notes:	For Combo Boxes and List Boxes, sets the text of the item specified by <index>. The first row is number 1, not 0.

Show	
Syntax:	Show(<Elementname>)
Parameters:	
<Elementname>	The name of the element to show.
Returns:	Nothing
Examples:	Show("prompt.element")
Notes:	Shows a multiprompt element that had been hidden. Elements are referenced using the syntax “prompt.element”.

6.4 File Functions

FileAppend	
Syntax:	FileAppend(<Filename> , <String>, [AllowAnyPath])
Parameters:	
<Filename>	File to append to
<String>	Data to be appended to file
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Nothing
Examples:	FileAppend("File.txt" , "This is a test" & ascCR & ascLF)
Notes:	Appends the data in <String> to the file named <Filename>. CR/LF are not appended automatically. You can append them using the constants ascCR and ascLF.

FileClose	
Syntax:	FileClose(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	The File to close
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Nothing
Examples:	FileClose("File.txt")
Notes:	Closes the file named by <filename> if it had been opened by FileOpen.

FileCopy	
Syntax:	FileCopy(<sourcefile>, <destfile>, <overwrite>, [AllowAnyPath])
Parameters:	
<sourcefile>	Original file to copy.
<destfile>	Name of the destination file to create.
<overwrite>	1 to overwrite an existing destination file, 0 to fail if the destination file exists.
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	This function returns 1 if the file copy succeeds, or 0 if it fails.
Examples:	FileCreate("File.txt")
Notes:	Copies the file named in <sourcefile> to <destfile>. Overwrites an existing <destfile> if the <overwrite> parameter is not zero, otherwise this function fails if the destination file already exists.

FileCreate	
Syntax:	FileCreate(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to create
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Nothing
Examples:	FileCreate("File.txt")
Notes:	Creates an empty file named <Filename>.

FileDate	
Syntax:	FileDate(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File whose date should be retrieved.
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	The last modification date/time of the file.
Examples:	@ret@ = FileDate("Text.txt")
Notes:	The file must be in the ITB directory. If the file is not found, an empty string is returned. If the file is found, the date/time will be returned in the format MM/DD/YYYY HH:MM:SS.

FileDelete	
Syntax:	FileExists(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	The file name to check.
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Returns 1 if the file exists, 0 if not.
Examples:	@ret@ = FileExists("Text.txt")
Notes:	The file must be located on the ITB directory.

FileEOF	
Syntax:	FileEOF(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to check
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	1 if the read position of the file is at the End Of File, or 0 is not.
Examples:	FileDelete("Text.txt")
Notes:	The file must have been opened with FileOpen.

FileExists	
Syntax:	FileExists(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to delete
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Nothing
Examples:	FileDelete("Text.txt")
Notes:	Deletes the file named by <Filename>.

FileList	
Syntax:	FileList(<List>, <Filespec>)
Parameters:	
<List>	The List to receive the list of files matching the filespec.
<Filespec>	The file specification to search for. Example: *.jpg
Returns:	Nothing
Examples:	FileList("FileList", "*.txt")
Notes:	The lists will be add to the list with the file number as the key, and the file name as the Value. The files will be listed from the ITB directory.

FileOpen	
Syntax:	FileOpen(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to open
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Returns 1 if the file was opened, or 0 if it failed.
Examples:	FileDelete("Text.txt")
Notes:	Open the file named by <filename> for reading. Use the FileReadLine function to read data from the file.

FilePosition	
Syntax:	FilePosition(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to position
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	The current file position
Examples:	FilePosition("Text.txt")
Notes:	Returns the current read position in the file named by <filename>. This value can be later used in the FileSeek function to reposition the Read position.

FileRead	
Syntax:	FileRead(<Filename>, <Length>, [AllowAnyPath])
Parameters:	
<Filename>	File to read
<Length>	The number of characters to read.
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	The text read from the file
Examples:	FileRead("Text.txt", 50)
Notes:	Reads text from the file.

FileReadLine	
Syntax:	FileReadLine(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to read
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	The text read from the file
Examples:	FileReadLine("Text.txt")
Notes:	Reads a line of text from the file named by <filename>. Text lines must be terminated by a \n (New Line) characters. Carriage Return characters are ignored. The Carriage Return and New Line characters are not returned with the data.

FileRename	
Syntax:	FileRename(<Oldname>, <Newname>, [AllowAnyPath])
Parameters:	
<Oldname>	File to rename
<Newname>	New name for file
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Nothing
Examples:	FileRename("OldFile.txt",”NewFile.txt”)
Notes:	Renames the file named by <Filename> to <Newname>.

FileSeek	
Syntax:	FileSeek(<Filename>, <Position>, [AllowAnyPath])
Parameters:	
<Filename>	File to seek
<Position>	The position in the file to seek to.
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Nothing
Examples:	FileSeek("Text.txt", @position@)
Notes:	Sets the current read position of the file named by <filename> to <position>. The position should have been retrieved with the FileGetPosition function.

FileSize	
Syntax:	FileSize(<Filename>, [AllowAnyPath])
Parameters:	
<Filename>	File to check
[AllowAnyPath]	Optional. If not specified or 0, any path provided in the filename will be remove and replaced with the ITB path. If 1, any path provided will be used.
Returns:	Returns the current size of the file named by <filename>.
Examples:	FileSize("Text.txt")
Notes:	

UnzipArchive	
Syntax:	UnzipArchive(<archive>)
Parameters:	
<archive>	The archive file to unzip.
Returns:	1 if the unzip was succesful, or 0 if it failed.
Examples:	FileSize("Text.txt")
Notes:	Unzips all of the files from the ZIP archive file. The archive must be located in the ITB directory, and the files will also be unzipped into the ITB directory. Existing files will be overwritten.

UnzipFile	
Syntax:	UnzipFile(<archive>, <fileinarchive>, [fileondevice])
Parameters:	
<archive>	The archive file to unzip.
<fileinarchive>	The filename of the file in the zip archive to unzip.
[fileondevice]	The filename of the new file on the device.
Returns:	1 if the unzip was succesful, or 0 if it failed.
Examples:	FileSize("Text.txt")
Notes:	If [fileondevice] is not specified, the <fileinarchive> will be used for both the device file and the filename in the archive. The archive must be located in the ITB directory, and the files will also be unzipped into the ITB directory. Existing files will be overwritten.

ZipFile	
Syntax:	ZipFile(<archive>, <fileinarchive>, [fileondevice])
Parameters:	
<archive>	The archive file to add the file into.
<fileinarchive>	The filename of the new file in the zip archive.
[fileondevice]	The filename of the file on the device.
Returns:	1 if the zip was succesful, or 0 if it failed.
Examples:	FileSize("Text.txt")
Notes:	If [fileondevice] is not specified, the <fileinarchive> will be used for both the device file and the filename in the archive. The archive must be located in the ITB directory, and the files will also be unzipped into the ITB directory. Existing files will be overwritten.

6.5 GPS Functions

GPSClose	
Syntax:	GPSClose( )
Parameters:	None
Returns:	Nothing
Examples:	GPSClose( )
Notes:	Closes the connection to the GPS device. This function works only on Windows Mobile 5 and higher devices that support the Microsoft GPS Intermediate Driver.

GPSDistance	
Syntax:	GPSDistance(<latitude1>, <longitude1>, <latitude2>, <longitude2>)
Parameters:	
<latitude1>	The Latitude of point 1.
<longitude1>	The Longitude of point 1.
<latitude2>	The Latitude of point 2.
<longitude2>	The Longitude of point 2.
Returns:	The approximate distance between the points, in US miles.
Examples:	@distance@ = GPSDistance(38.889262, -77.04978, 38.881212, -77.036476)
Notes:	Calculates an approximate distance between the two latitude/longitude pairs using the Haversine formula.Returns the distance in US miles.

GPSGetPosition	
Syntax:	GPSGetPosition(<maxage>, <latitude>, <longitude>, <speed>, <heading>, <altitude>, <numsatellites>)
Parameters:	
<maxage>	The maximum age of valid data in seconds.
<latitude>	A variable to receive the latitude.
<longitude>	A variable to receive the longitude.
<speed>	A variable to receive the speed.
<heading>	A variable to receive the heading.
<altitude>	A variable to receive the altitude.
<numsatellites>	A variable to receive the number of satellites.
Returns:	A value indicating the type of fix. 0 indicates an error or no fix. 2 indicates a 2-D fix. 3 indicates a full 3-D fix.
Examples:	@ret@ = GPSGetPosition(30, @lat@, @long@, @speed@, @heading@, @altitude@, @numsat@)
Notes:	The GPS device should have been previously opened with GPSOpen. This function works only on Windows Mobile 5 and higher devices that support the Microsoft GPS Intermediate Driver. If the GPS device does not report current information for a particular parameter, it will use the latest valid data it had, and prepend an asterisk. For example, *80.1234.

GPSIsOpen	
Syntax:	GPSIsOpen( )
Parameters:	None
Returns:	1 if the device is connected and operating, 0 if not.
Examples:	@ret@ = GPSIsOpen( )
Notes:	Returns the status of the GPS hardware and driver. The GPS device should have been previously opened with GPSOpen. This function works only on Windows Mobile 5 and higher devices that support the Microsoft GPS Intermediate Driver.

GPSOpen	
Syntax:	GPSOpen( )
Parameters:	None
Returns:	Nothing
Examples:	GPSOpen( )
Notes:	Establishes a connection tot he GPS device. This function must be called before the GPSGetPosition function will work. It can take several minutes for the GPS hardware to establish a fix once it has been enabled using thsi function. The device must be configured in the Control Panel GPS applet. This function works only on Windows Mobile 5 and higher devices that support the Microsoft GPS Intermediate Driver.

GPSSetUserField	
Syntax:	GPSSetUserField(<FieldValue>)
Parameters:	
<FieldValue>	The user-defined value to be stored with the GPS Tracking data.
Returns:	Nothing
Examples:	GPSSetUserField("1234")
Notes:	This function allows you to store an additional User-Defined value with your GPS tracking data.

GPSTrackingParameters

Syntax:	GPSTrackingParameters(<gpsmode>, <filtering>, <separatefile>, <sendrate>, <usegprs>, <connectionname>, <leaveopen>)
Parameters:	
<gpsmode>	The GPS Tracking mode to set. 0=Leave GPS mode unchanged. 1=GPS on, but not collecting data. 2=Collect tracking data.
<filtering>	The Filtering to use. 0=More Precise, 1=Average, 2=Less Data.
<separatefile>	Whether to collect data into a program-specific file. 1=program-specific file, 0=default file.
<sendrate>	How often should the data be sent to the OMNI Server. 0=manual send, otherwise specify the time in seconds.
<usegprs>	Set whether to use GPRS to send the data to the OMNI Server. 1=yes, 0=no.
<connectionname>	If using GPRS, sets the RAS connection name to connect.
<leaveopen>	Sets whether to leave the GPRS connection open after sending. 1=yes, 0=no.
Returns:	Nothing
Examples:	GPSTrackingParameters(2, 0, 0, 60, 1, "GPRS", 1)
Notes:	This function allows you to override the GPS Tracking settings for the program.

6.6 Logical Functions

And	
Syntax:	AND(<Expression1> , <Expression2>)
Parameters:	
<Expression1>	Logical expression to be evaluated
<Expression2>	Logical expression to be evaluated
Returns:	Returns 1 (same asTRUE) if <Expression1> and <Expression2> are both non-zero, else 0 (same as FALSE).
Examples:	AND(IsNumeric("5"), IsNumeric("7")) returns 1 (TRUE)
Notes:	AND is a function, not an operator. The parameters should evaluate to logical expressions. Before testing, VAL() will be performed on each expression. Any non-numeric value is considered FALSE. Any non-zero Numeric value is considered TRUE. Values starting with numeric digits will be converted up to the first non-numeric character.

IIF	
Syntax:	IIF(<Expression> , <Result1> , <Result2>)
Parameters:	
<Expression>	Logical expression to be evaluated
<Result1>	The expression to be returned if <Expression> is true
<Result2>	The expression to be returned if <Expression> is false
Returns:	Returns <Result1> if <Expression> is true, else returns <Result2>
Examples:	IIF(IsNumeric(@expr@),"Number","Alpha") returns "Number" if @expr@ evaluates to a numeric expression, otherwise "Alpha" is returned.
Notes:	The <Expression> should evaluate to a logical expression. Before testing, VAL() will be performed on <Expression>. Any non-numeric value is considered FALSE. Any non-zero Numeric value is considered TRUE. Values starting with numeric digits will be converted up to the first non-numeric character.

Not	
Syntax:	NOT(<Expression>)
Parameters:	
<Expression>	Logical expression to be evaluated.
Returns:	Returns 0 (same as FALSE) if <Expression> is non-zero, else 1 (same as TRUE).
Examples:	NOT(IsNumeric("5")) returns 0 (FALSE)
Notes:	NOT is a function, not an operator. The parameter should evaluate to a logical expression. Before testing, VAL() will be performed on the expression. Any non-numeric value is considered FALSE. Any non-zero Numeric value is considered TRUE. Values starting with numeric digits will be converted up to the first non-numeric character.

Or	
Syntax:	OR(<Expression1>, <Expression2>)
Parameters:	
<Expression1>	Logical expression to be evaluated.
<Expression2>	Logical expression to be evaluated.
Returns:	Returns 1 (same as TRUE) if either <Expression1> or <Expression2> are non-zero, else 0 (same as FALSE).
Examples:	OR(IsNumeric("A"), IsNumeric("7")) returns TRUE
Notes:	OR is a function, not an operator. The parameters should evaluate to logical expressions. Before testing, VAL() will be performed on each expression. Any non-numeric value is considered FALSE. Any non-zero Numeric value is considered TRUE. Values starting with numeric digits will be converted up to the first non-numeric character.

6.7 Lookup Functions

AddValidation	
Syntax:	AddValidation(<file>, <Field1>, <Value1>, ...)
Parameters:	
<file>	The validation file to add the record to.
<Field1>	The field name of a field to set
<Value1>	The value of the field to set
…	You may specify any number of Field/Value pairs
Returns:	Nothing
Examples:	AddValidation("val.txt", "description", "123456", "sku", "12345")
Notes:	Adds a record to a validation file. The filename without path should be in <file>. The fields to insert are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. The validation file must have been defined in the Validation Files screen.

AddValidationList	
Syntax:	AddValidationList(<file>, <listname>)
Parameters:	
<file>	The validation file to add the record to.
<listname>	The name of a list containing the data to insert into the validation file.
Returns:	Nothing
Examples:	AddValidationList("val.txt", "listname")
Notes:	Adds a record to a validation file. The fields/values to insert should be given in the List named <listname>. The validation file must have been defined in the Validation Files screen.

CountCollect	
Syntax:	CountCollect(<Field1>, <Value1>, ...)
Parameters:	
<Field1>	Prompt or element name of the field in the collected data file
<Value1>	Value to match on
…	You may specify up to 5 Field/Value pairs
Returns:	Returns the number of matching records
Examples:	CountCollect(“sku”, “12345”) returns the number of records in the collected data file where the data collected for the prompt named “sku” is “12345”.
Notes:	Counts the number of records in the collected data file matching the specified criteria. If no match fields are specified, the total count of all collected records is returned.

CountValidation	
Syntax:	CountValidation(<Filename>, <Field1>, <Value1>, ...)
Parameters:	
<Filename>	The validation file name to use.
<Field1>	Prompt or element name of the field in the collected data file
<Value1>	Value to match on
…	You may specify up to 5 Field/Value pairs
Returns:	Returns the number of matching records
Examples:	CountValidation("samepl.csv", “sku”, “12345”) returns the number of records in the validation file where the data for the field named “sku” is “12345”.
Notes:	Counts the number of records in the validation file matching the specified criteria. If no match fields are specified, the total count of all records is returned.

DeleteCollect	
Syntax:	DeleteCollect (<all> , <Field1>, <Value1>, ...)
Parameters:	
<all>	Pass “1” to delete all matching records, or “0” to delete just the first match.
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
…	Up to 5 Field/Value pairs may be specified
Returns:	Nothing
Examples:	DeleteCollect(“1”, “sku”, “12345”) deletes all collected data records where the value of the “sku” prompt is “12345”.
Notes:	At least one match field and value must be specified.

DeleteValidation	
Syntax:	DeleteValidation (<all> , <Field1>, <Value1>, ...)
Parameters:	
<all>	Pass “1” to delete all matching records, or “0” to delete just the first match.
<Field1>	The prompt or element name of the field in the validation file to match
<Value1>	The value to match
…	Any number of Field/Value pairs may be specified
Returns:	The number of records deleted
Examples:	@Count@ = DeleteValidation("val.txt", "sku", "12345")
Notes:	Deletes records from a validation file. The filename without path should be in <file>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

ExecuteSQLite	
Syntax:	ExecuteSQLite (<databasefile>, <sql>)
Parameters:	
<databasefile>	The SQLite database file.
<sql>	The SQL Statement to execute.
Returns:	The result of the query.
Examples:	@ret@ = ExecuteSQLite("ValFile1.csv.dbval", "Select sum(Field1) from Table1")
Notes:	This function can return only one row. If more than one row is the result of the query, only the first row will be returned. If more than one field are returned, the fields will be tab-delimited.

ExecuteSQLiteTable	
Syntax:	ExecuteSQLite (<databasefile>, <sql>, <tablename>)
Parameters:	
<databasefile>	The SQLite database file.
<sql>	The SQL Statement to execute.
<tablename>	The table to receive the results
Returns:	Nothing
Examples:	@ret@ = ExecuteSQLiteTable("ValFile1.csv.dbval", "Select Field1, Field2, Field3 from Table1", "ResultTable")
Notes:	Executes a SQL Statement against the SQLite database. Returns the result of the query in the table named <tablename>.

InitializeValidation	
Syntax:	InitializeValidation (<filename>, [InitIfExists])
Parameters:	
<filename>	The name of the Validation File to initialize.
[InitIfExists]	Whether to initialize the file even if it exists.
Returns:	1 if the file was created, or 0 if is was not.
Examples:	@ret@ = InitializeValidation("ValFile.csv", 1)
Notes:	Creates and initializes a validation file table in Sqlite. The filename without path should be in <file>. The validation file must have been defined in the Validation Files screen. The [InitIfExists] parameter controls what happens if the Validation File already exists. If [InitIfExists] in non-zero, any existing file will be deleted and recreated. If [InitIfExists] is zero, the file will only be created if it does not exist.

InsertFileToValidationBinaryField	
Syntax:	InsertFileToValidationBinaryField (<filename>, <binaryfile>, <binaryfield>, <Field1>, <Value1>, ...)
Parameters:	
<filename>	The name of the Validation File to initialize.
<binaryfile>	The filename of the ibnary file to insert.
<binaryfield>	The binary field in the validation file.
<Field1>	Match field name.
<Value1>	Match field value.
Returns:	Nothing
Examples:	InsertFileToValidationBinaryField("val.txt", "newimage.jpg", "image", "sku", "12345")
Notes:	Inserts the contents of a file into a binary field in a validation file. The validation filename without path should be in <file>. The filename without path should be in <binaryfile>. The field to update should be given in <binaryfile>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen, and the <binaryfield> field must be a binary field.

LastCollect	
Syntax:	LastCollect(<Lookup>)
Parameters:	
<Lookup>	The prompt or element name of the field to retrieve
Returns:	Returns the last value collected for the prompt or element named <Lookup>
Examples:	LastCollect(“sku”) returns the last value collected for the prompt named “sku”.
Notes:	This function returns the last data collected.

LastCollectList	
Syntax:	LastCollectList(<ListName>)
Parameters:	
<ListName>	The name of the list to receive the collected data record
Returns:	1 if a record was found, or 0 if not.
Examples:	@ret@ = LastCollectList("TestList")
Notes:	This function returns the last data collected in the list specified.

LastCollectRecord	
Syntax:	LastCollectRecord( )
Parameters:	None
Returns:	Returns the last collected data record.
Examples:	@record@ = LastCollectRecord() returns the last record collected.
Notes:	This function returns the last data collected.

LookupCollect	
Syntax:	LookupCollect(<Lookup>, <Field1>, <Value1>, ...)
Parameters:	
<Lookup>	The prompt or element name of the field in the collected data file to lookup
<Field1>	The prompt or element name of a field in the collected data file to match
<Value1>	The data to match
…	Up to 5 Field/Value pairs may be specified
Returns:	Returns the value of a field in the collected data file
Examples:	LookupCollect(“qty”, “sku”, “12345”) returns the data collected for the prompt named “qty” where the value collected for the field named “sku” was “12345”.
Notes:	Performs a lookup from the collected data file. <Lookup> is the prompt or element name of the field to lookup. <Field1>, <Field2>, etc specify the prompt or element name of a field to match. <Value1>, <Value2>, etc specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, only the first collected (oldest) record will be returned.

LookupCollectList	
Syntax:	LookupCollectList(<Listname>, <Field1>, <Value1>, ...)
Parameters:	
<Listname>	The name of the list to receive the collected data record
<Field1>	The prompt or element name of a field in the collected data file to match
<Value1>	The data to match
Returns:	Returns the entire collected data record matching the field / value pairs on the list specified.
Examples:	LookupCollectList("ListName", "item.sku", "12345")
Notes:	Performs a lookup of a record from the collected data file and places it into a List. The List name is <listname>. <Field1>, <Field2>, etc specify the prompt name of a field to match. <Value1>, <Value2>, etc specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, only the first record will be returned.

LookupCollectRecord	
Syntax:	LookupCollectRecord(<Field1>, <Value1>, ...)
Parameters:	
<Field1>	The prompt or element name of a field in the collected data file to match
<Value1>	The data to match
Returns:	Returns the entire collected data record matching the field / value pairs.
Examples:	@record@ = LookupCollectRecord(“sku”, “12345”) returns the collected data record where the value collected for the field named “sku” was “12345”.
Notes:	Performs a lookup from the collected data file. <Field1>, <Field2>, etc specify the prompt or element name of a field to match. <Value1>, <Value2>, etc specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, only the oldest first collected (oldest) record will be returned.

LookupCollectReverse	
Syntax:	LookupCollectReverse(<Lookup>, <Field1>, <Value1>, ...)
Parameters:	
<Lookup>	The prompt or element name of the field in the collected data file to lookup
<Field1>	The prompt or element name of a field in the collected data file to match
<Value1>	The data to match
Returns:	Returns the value of a field in the collected data file
Examples:	LookupCollectReverse(“qty”, “sku”, “12345”) returns the data collected for the prompt named “qty” where the value collected for the field named “sku” was “12345”.
Notes:	Performs a lookup from the collected data file. <Lookup> is the prompt or element name of the field to lookup. <Field1>, <Field2>, etc. specify the prompt or element name of a field to match. <Value1>, <Value2>, etc. specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, only the last collected (most recent) record will be returned.

LookupCollectReverseList	
Syntax:	LookupCollectReverseList(<Listname>, <Lookup>, <Field1>, <Value1>, ...)
Parameters:	
<Listname>	The name of a list to receive the collected data.
<Field1>	The prompt or element name of a field in the collected data file to match.
<Value1>	The data to match
Returns:	Nothing
Examples:	LookupCollectReverseList("ListName", "item.sku", "12345")
Notes:	Performs a reverse lookup of a record from the collected data file and places it into a List. The List name is <listname>. <Field1>, <Field2>, etc specify the prompt name of a field to match. <Value1>, <Value2>, etc specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, only the first record will be returned.

LookupCollectReverseRecord	
Syntax:	LookupCollectReverseRecord(<Field1>, <Value1>, ...)
Parameters:	
<Field1>	The prompt or element name of a field in the collected data file to match
<Value1>	The data to match
…	Up to 5 Field/Value pairs may be specified
Returns:	Returns the last collected data record matching the field/value pairs.
Examples:	@record@ = LookupCollectReverseRecord(“sku”, “12345”) returns the data collected where the value collected for the field named “sku” was “12345”.
Notes:	Performs a lookup from the collected data file. <Field1>, <Field2>, etc. specify the prompt or element name of a field to match. <Value1>, <Value2>, etc. specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, only the last collected (most recent) record will be returned.

LookupCollectTable	
Syntax:	LookupCollectTable(<Tablename>, <Field1>, <Value1>, ...)
Parameters:	
<Tablename>	The name of the table to receive the records.
<Field1>	The prompt or element name of a field in the collected data file to match
<Value1>	The data to match
Returns:	Return the number of records returned in the table.
Examples:	LookupCollectTable("TableName", "item.sku", "12345")
Notes:	Performs a lookup of any number of records from the collected data file and places them into a Table. The Table name is <tablename>. <Field1>, <Field2>, etc specify the prompt name of a field to match. <Value1>, <Value2>, etc specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. If more than one record matches the criteria specified, all of the records will be placed in the table.

LookupParseCollectField	
Syntax:	LookupParseCollectField(<Field>, <Record>)
Parameters:	
<Field>	The field in the collected data record to return
<Record>	A stored record from the collected data file previously retrieved with the LookupCollectRecord function
Returns:	The data from the field <field> in the stored record
Examples:	LookupParseCollectField(“Prompt1.Textbox”, @Record@) returns the value in the “Prompt1.Textbox” field from the record stored in the user variable @Record@
Notes:	Parses a field from a collected data record. The field to parse should be given in <Field>. The data record should be in <Record> and be the result of the LookupCollectRecord or LookupCollectRecordReverse function.

LookupParseValidationField	
Syntax:	LookupParseValidationField(<File>, <Field>, <Record>)
Parameters:	
<File>	The validation file to use for the lookup
<Field>	The field in the validation file to return
<Record>	A stored record from the validation file previously retrieved with the LookupValidationRecord function
Returns:	The data from the field <field> in the stored record
Examples:	LookupParseValidationField(“val.txt”, “description”, @Record@) returns the value in the “description” field of the validation file “val.txt”, from the record stored in the user variable @Record@
Notes:	Parses a field from a validation file record. The filename without path should be in <File>. The validation file must have been defined in the Validation Files screen, even if it is not being used on the Advanced Prompt Settings screen. The field to parse should be given in <Field>. The data record should be in <Record> and be the result of the LookupValidationRecord function.

LookupValidation	
Syntax:	LookupValidation(<File>, <Lookup>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to use for the lookup
<Lookup>	The field in the validation file to return
<Field1>	The field to match
<Value1>	The value to match
Returns:	The data from the field <Lookup> in the record matching the Field/Value criteria
Examples:	LookupValidation(“val.txt”, “description”, “sku”, “12345”) returns the value in the “description” field of the validation file “val.txt”, from the record where the “sku” field is equal to “12345”.
Notes:	Performs a lookup from a validation file. The filename without path should be in <File>. The field to lookup should be given in <Lookup>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen, even if it is not being used on the Advanced Prompt Settings screen.

LookupValidationList	
Syntax:	LookupValidationList(<File>, <ListName>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to use for the lookup
<ListName>	The name of a list to receive the data.
<Field1>	The field to match
<Value1>	The value to match
Returns:	Returns 1 if a record was found, or 0 if not.
Examples:	@ret@ = LookupValidationList("val.txt", "ListName", "sku", "12345")
Notes:	Performs a lookup of a record from a validation file into a List. The filename without path should be in <file>. The List name to receive the results should be given in <listname>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

LookupValidationRecord	
Syntax:	LookupValidationRecord(<File>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to use for the lookup
<Field1>	The field to match
<Value1>	The value to match
Returns:	The record matching the Field/Value criteria
Examples:	@Record@ = LookupValidationRecord("val.txt", "sku", "12345") returns the record (all the fields) from the validation file “val.txt” where the “sku” field is equal to “12345”.
Notes:	Performs a lookup from a validation file and returns the entire record. The filename without path should be in <file>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen, even if it is not being used on the Advanced Prompt Settings screen. Use the LookupParseValidationField function to parse individual fields from the returned record.

LookupValidationTable	
Syntax:	LookupValidationTable(<File>, <TableName>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to use for the lookup
<TableName>	The name of a table to receive the data.
<Field1>	The field to match
<Value1>	The value to match
Returns:	Returns 1 if a record was found, or 0 if not.
Examples:	@ret@ = LookupValidationTable("val.txt", "TableName", "sku", "12345")
Notes:	Performs a lookup of a group of records from a validation file, and places them into a Table. The filename without path should be in <file>. The table to receive the results should be given in <tablename>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

PickListField	
Syntax:	PickListField(<Field>)
Parameters:	
<Field>	The name of the field to return
Returns:	The data from a field in a validation file
Examples:	PickListField(“descr”) returns the value of the “descr” field in the validation file for this record.
Notes:	This function is used only in the Override Display Prompt In-Prompt Script. As each record is added to the picklist, the Override Display Prompt In-Prompt Script is called. You can format the text to be displayed in the picklist. This function can be used to retrieve the value of a validation file field.

SaveCollectedData	
Syntax:	SaveCollectedData(<validate>)
Parameters:	
<validate>	Whether to validate and save the current prompt before saving the record.
Returns:	Nothing
Examples:	SaveCollectedData (1) saves the current values of all prompts to the collected data file. SaveCollectedData (0) saves a record but does not validate or save the changes made on the current prompt.
Notes:	This function causes a collected data record to be written, using the current values of all prompts and elements. This is exactly like accepting the last prompt in the program. This function is only supported on Windows CE and PocketPC clients, and the PC Client. It is not supported on DOS devices.

SaveValidationBinaryFieldToFile	
Syntax:	SaveValidationBinaryFieldToFile(<file>, <lookup>, <savefile>, <Field1>, <Value1>, ...)
Parameters:	
<file>	The validation file containing the binary data.
<lookup>	The field in the validation file to lookup.
<savefile>	The storage file to save the binary data into.
<Field1>	The field to match
<Value1>	The value to match
Returns:	Nothing
Examples:	SaveValidationBinaryFieldToFile("val.txt", "image", "saveimage.jpg", "sku", "12345")
Notes:	Performs a lookup from a validation file and saves a binary field to a file. The filename without path should be in <file>. The field to lookup should be given in <lookup>. The save file should be given in <savefile>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

SumCollect	
Syntax:	SumCollect (<Lookup> , <Field1>, <Value1>, ...)
Parameters:	
<Lookup>	The prompt or element name of the field in the collected data file to sum
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
…	Up to 5 Field/Value pairs may be specified
Returns:	Returns the sum of the specified field matching the criteria
Examples:	SumCollect(“qty”, “sku”, “12345”) returns the sum of the data collected for the “qty” field in the collected data file where the data collected for the prompt named “sku” is “12345”.
Notes:	Sums the values in the collected data file matching the specified criteria. <Lookup> specifies the prompt or element name of the field in the collected data file to sum. <Field1>, <Field2> etc specify the prompt or element name of the field in the collected data file to match. <Value1>, <Value2> etc specify the values to match. If no match fields are specified, the sum for all collected records is returned.

SumValidation	
Syntax:	SumValidation (<Filename>, <Lookup> , <Field1>, <Value1>, ...)
Parameters:	
<Filename>	The file name of the validation file to use.
<Lookup>	The prompt or element name of the field in the collected data file to sum
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
…	Up to 5 Field/Value pairs may be specified
Returns:	Returns the sum of the specified field matching the criteria
Examples:	SumValidation("Sample.csv", “qty”, “sku”, “12345”) returns the sum of the data for the “qty” field in the validation file where the data for the field named “sku” is “12345”.
Notes:	Sums the values in the validation file matching the specified criteria. If no match fields are specified, the sum for all records is returned.

UpdateCollect	
Syntax:	UpdateCollect (<Field>, <Data>, <All> , <Field1>, <Value1>, ...)
Parameters:	
<Field>	The name of the prompt or element in the collected data to update
<Data>	The data to store as the new value.
<All>	Pass “1” to update all matching records, or “0” to update just the first match.
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
Returns:	Nothing
Examples:	UpdateCollect(“qty”, “10”, “1”, “sku”, “12345”) updates all collected data records where the value of the “sku” prompt is “12345”, and sets the “qty” field to “10”.
Notes:	At least one Match field and Value must be specified.

UpdateCollectField	
Syntax:	UpdateCollectField (<record> , <field>, <data>)
Parameters:	
<record>	The collected data record.
<field>	The name of the field to update.
<data>	The data to place into the field.
Returns:	Nothing
Examples:	UpdateCollect(@record@, “qty”, “10”) updates the collected data record in @record@, setting the “qty” field to “10”.
Notes:	This function updates a collected data record in a string variable. This function is used in conjunction with LookupCollectRecord or LookupCollectRecordReverse to allow changing the values of any number of fields on a collected data record. You must call UpdateCollectRecord to write the record back to the collected data file.

UpdateCollectList	
Syntax:	UpdateCollectList (<Listname>, <All> , <Field1>, <Value1>, ...)
Parameters:	
<Listname>	The name of the list containing the data to update into the validation file.
<All>	Pass “1” to update all matching records, or “0” to update just the first match.
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
Returns:	Nothing
Examples:	UpdateCollectList("ListName", 0, "prompt.qty", "10")
Notes:	Writes an updated collected data record to the collected data file. The collected data record fields and values are in <listname>. <Field1>, <Field2>, etc specify the prompt name of a field to match. <Value1>, <Value2>, etc specify the values of those fields in the collected data. Up to 5 match fields can be specified in this way. <all> indicates if all matching records should be updated, or only the first one.

UpdateCollectRecord	
Syntax:	UpdateCollectRecord (<record>, <All> , <Field1>, <Value1>, ...)
Parameters:	
<record>	The collected data in a string variable.
<All>	Pass “1” to update all matching records, or “0” to update just the first match.
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
Returns:	Nothing
Examples:	UpdateCollectRecord(@record@, “1”, “sku”, “12345”) updates the collected data records where the value of the “sku” prompt is “12345”.
Notes:	At least one Match field and Value must be specified. After retrieving a collected data record into a string with LookupCollectRecord or LookupCollectRecordReverse, use UpdateCollectField to change the values. Finally, use UpdateCollectRecord to save those changes back to the collected data file.

UpdateValidation	
Syntax:	UpdateValidation (<File>, <UpdateField>, <UpdateValue>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to update.
<UpdateField>	The field in the validation file to update.
<UpdateValue>	The value to update into the validation file.
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
Returns:	the number of records updated.
Examples:	UpdateValidation("val.txt", "description", "123456", "sku", "12345")
Notes:	Updates records in a validation file. The filename without path should be in <file>. The field to update should be given in <updatefield>, and the value to put in that field should be in <updatevalue>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

UpdateValidationField	
Syntax:	UpdateValidationField (<File>, <Record>, <Field>, <Value>)
Parameters:	
<File>	The validation file to update.
<Record>	The field in the validation file record to update.
<Field>	The prompt or element name of the field in the validation file data to match.
<Value>	The value to match.
Returns:	Nothing
Examples:	UpdateValidationField("val.txt", @record@, "sku", "12345")
Notes:	Updates a field in a validation file record. This record should have been retrieved with LookupValidationRecord. The filename without path should be in <file>. The record to update is in <record>. The new value for the field <field> should be given in <value>. The validation file must have been defined in the Validation Files screen.

UpdateValidationList	
Syntax:	UpdateValidation (<File>, <ListName>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to update.
<ListName>	The name of the list containing the data to update into the validation file.
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
Returns:	The number of records updated.
Examples:	UpdateValidationList("val.txt", "listname", "sku", "12345")
Notes:	Updates records in a validation file. The filename without path should be in <file>. The fields/values to update should be given in the List named <listname>. This List should have been retrieved with a call to LookupValidationList. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

UpdateValidationRecord	
Syntax:	UpdateValidationRecord (<File>, <Record>, <Field1>, <Value1>, ...)
Parameters:	
<File>	The validation file to update.
<Record>	The string containing the Validation file data
<Field1>	The prompt or element name of the field in the collected data file to match
<Value1>	The value to match
Returns:	The number of records updated.
Examples:	UpdateValidationRecord("val.txt", @record@, "sku", "12345")
Notes:	Updates records in a validation file. The filename without path should be in <file>. The new values for the fields should be given in <record>. This record must be the result of a call to LookupValidationRecord, updated with calls to UpdateValidationField. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

6.8 Math Functions

Abs	
Syntax:	Abs(<Value>)
Parameters:	
<Value>	A numeric value to be converted
Returns:	Returns the absolute value of <Value>, ignoring sign.
Examples:	Abs(3.5) = 3.5 Abs(-3) = 3
Notes:	Conversion stops at the first non-numeric character. Numeric characters are digits, '-', and '.'

Fix	
Syntax:	Fix(<Value>)
Parameters:	
<Value>	A number to be converted to an integer
Returns:	The integer portion of <Value>
Examples:	Fix(3.5) = 3, Fix(-3.8) = -3
Notes:	Removes the fractional part of <Value> and returns the resulting integer value. The largest number supported is +/- 2147483648.

Int	
Syntax:	Int(<Value>)
Parameters:	
<Value>	A number to be converted to an integer
Returns:	Returns the largest integer which is less than or equal to <Value>
Examples:	Int(3.5) = 3, Int(-3.2) = -4
Notes:	Use Int when you want the number to be less than <Value>. Use Fix to truncate the fractional portion. For positive numbers, Int and Fix return the same result. For negative numbers, they do not. The largest number supported is +/- 2147483648.

Mod	
Syntax:	Mod(<Value1> , <Value2>)
Parameters:	
<Value1>	The numerator for the division.
<Value2>	The denominator for the division.
Returns:	Returns the remainder of the division of <Value1> by <Value2>
Examples:	Mod(6,2) = 0 Mod(11,3) = 2
Notes:	The expressions <Value1> and <Value2> will be converted to integer data. The conversion stops at the first non-integer character. The largest number supported is +/- 2147483648.

Quotient	
Syntax:	Quotient(<Value1> , <Value2>)
Parameters:	
<Value1>	An integer used for the numerator
<Value2>	An integer used for the denominator
Returns:	Returns the integer results of the division of <Value1> by <Value2>
Examples:	Quotient(6,2) = 3 Quotient(13,3) = 4
Notes:	The expressions <Value1> and <Value2> will be converted to integers. The expressions will be converted up to the first non-numeric character. The largest number supported is +/- 2147483648.

Rand	
Syntax:	Rand( )
Parameters:	None
Returns:	Returns a random number between 0 and 32767.
Examples:	Rand( )
Notes:	The random number generator is seeded on the first call to this function.

Round	
Syntax:	Round(<Value> , <Numplaces>)
Parameters:	
<Value>	A numeric value to be rounded
<Numplaces>	The number of decimal places to round to
Returns:	Rounds <Value> to the nearest decimal place indicated by <Numplaces>.
Examples:	Round(7257.28456,0) = 7257 Round(7257.28456,2) = 7257.28 Round(7257.28456,-2) = 7300
Notes:	The <Value> will be converted to numeric data. The conversion will stop at the first non-numeric character.

Sgn	
Syntax:	Sgn(<Value>)
Parameters:	
<Value>	The numeric data to test for sign
Returns:	-1 if <Value> is less than 0 0 if <Value> is equal to 0 1 if <Value> is greater than 0
Examples:	Sgn(3.5) = 1
Notes:	The <Value> will be converted to numeric data. The conversion will stop at the first non-numeric character.

Sqr	
Syntax:	Sqr(<Value>)
Parameters:	
<Value>	A numeric expression
Returns:	Returns the square root of <Value>
Examples:	Sqr(9) = 3 Sqr(8) = 2.828
Notes:	The <Value> will be converted to a numeric value. The conversion stops at the first non-numeric character. If a negative number is supplied, the square root of the absolute value will be taken.

6.9 Notification Functions

AllKeys	
Syntax:	AllKeys(<allkeys>)
Parameters:	
<allkeys>	Specify 1 to enable AllKeys mode, or 0 to disable it.
Returns:	Nothing
Examples:	AllKeys(1)
Notes:	Sets the data collection device into All Keys mode. This function only has effect on Windows CE and Windows Mobile devices which support the All Keys mode. Many Windows CE and Windows Mobile devices are pre-programmed with certain functions keys for system operations, such as Volume Control or the Notes or Recorder applet. This prevents those predefined keys from being available as hotkeys in ITScriptNet. Setting AllKeys (1) will configure the device to bring all keypresses to ITScriptNet, overriding the Windows defaults. Once the AllKeys function has been called, the device stays in the selected mode as long as the ITScriptNet client is running. The function only needs to be called once to set the mode. It does not need to be called again unless you want to change the mode to the other option. A good place to call the AllKeys function is in the Program Start event or in the Before Prompting event of the first prompt..

BackgroundProcess	
Syntax:	BackgroundProgress([variable1], [value1], ...)
Parameters:	
[variable1]	Optional. The name of a local variable to create in the Progress script.
[value1]	Optional. The value to pass for the local variable to the Progress script.
Returns:	Nothing
Examples:	BackgroundProgress("Variable1", "Value1")
Notes:	If called from within a Background Task script, causes the Progress event to be triggered. Has no effect if called from any other script. Any number of [variable], [value] pairs can be specified. These will be available in the Progress event as local variables named by the [variables], and with the value provided in each [value].

Beep	
Syntax:	Beep([async])
Parameters:	
[async]	Optional. Specifies whether the sound should be played synchronously (if zero - the script waits for the sound to be completed before returning) or asynchronously (if non-zero - the script continues while the sound is being played). If the parameter is not specified, the sound will be synchronous.
Returns:	Nothing
Examples:	Beep( )
Notes:	Causes the device to beep a high-pitched beep.

Break	
Syntax:	Break( )
Parameters:	Nonte
Returns:	Nothing
Examples:	Break( )
Notes:	Breaks into the debugger in the Simulator. Has no effect on a device client.

Buzz	
Syntax:	Buzz([async])
Parameters:	
[async]	Optional. Specifies whether the sound should be played synchronously (if zero - the script waits for the sound to be completed before returning) or asynchronously (if non-zero - the script continues while the sound is being played). If the parameter is not specified, the sound will be synchronous.
Returns:	Nothing
Examples:	Buzz( )
Notes:	Causes the device to buzz a low-pitched buzz.

DialogBox	
Syntax:	DialogBox(<Subprompt>, [Caption], [Layout])
Parameters:	
<Subprompt>	The name of the subprompt to display as a dialog box.
[Caption]	Optional. The caption to use for the dialog box.
[Layout]	Optional. The name of the subprompt layout to use for the dialog box.
Returns:	On Windows, the function returns 1 if the user presses the Accept button, and 0 if the user presses the Exit button. On Android the function returns nothing and does not block.
Examples:	DialogBox("SubPrompt1")
Notes:	Displays a subprompt to the user in a dialog box. The dialog is closed when the user presses either an Accept or Exit button. To return data from the dialog box to the calling prompt, use the AfterValidation event for the subprompt and assign the control values to Global Variables. The caption is optional. On Windows platforms, if a caption is specified, the dialog box will have a caption that allows the user to move the dialog box around on the screen. If the caption is blank or not specified, the dialog box will have no caption and cannot be moved. The layout is optional. You can specify the display name of the dialog subprompt layout (for example, "200x150") and the dialog will use that layout. If not specified, the client will select the first defined layout. The function returns 1 if the user presses the Accept button, and 0 if the user presses the Exit button. On Windows platforms, this function does not return until the dialog box has been closed. On Android, this function returns immediately.

ExitProgram	
Syntax:	ExitProgram( )
Parameters:	None
Returns:	Nothing
Examples:	ExitProgram( )
Notes:	Causes the data collection program to exit and return to the Client’s main menu. Supported for PocketPC Devices, CE Devices, and PC Client. Not supported on DOS Devices.

FlashLEDs	
Syntax:	FlashLEDs()
Parameters:	None
Returns:	Nothing
Examples:	FlashLEDs()
Notes:	Causes the device to flash the SCAN and DECODE LEDs for ½ second.

FlashPromptBackground	
Syntax:	FlashPromptBackground(<Color>, <Times>, <Interval>)
Parameters:	
<Color>	The color to use for the Prompt background when flashing.
<Times>	The number of times to flash.
<Interval>	The time interval in milliseconds between flashes.
Returns:	Nothing
Examples:	FlashPromptBackground(colorRed, 3, 1000)
Notes:	Flashes the background color of the current prompt, alternating between the <color> specified and the original color. The color will be changed to <color> and back the number of times specified in <times>, waiting the amount of time specified in <interval> in milliseconds. If the <times> parameter is 0 or negative, the prompt will flash until the program moves to a different prompt. If the program moves to a different prompt, the flashing is cancelled.

GetActiveSyncStatus	
Syntax:	GetActiveSyncStatus( )
Parameters:	none
Returns:	Returns whether there is an ActiveSync or Windows Device Manager connection established.
Examples:	@ret@ = GetActiveSyncStatus( )
Notes:	Returns 1 if the connection is active, or 0 if not. Only supported on Windows CE or PocketPC/Windows Mobile devices. Always returns 0 on the PC Client or Simulator.

GetBatteryLife	
Syntax:	GetBatteryLife( )
Parameters:	none
Returns:	Returns the remaining battery life.
Examples:	@ret@ = GetBatteryLife( )
Notes:	The battery life will be from 0% to 100%, or -1 if unknown.

GetKeyboardMode	
Syntax:	GetBatteryLife( )
Parameters:	none
Returns:	Returns the current keyboard mode.
Examples:	@ret@ = GetKeyboardMode( )
Notes:	The current Caps Lock or Num Lock state will be one of the following constants: keyNumericMode, keyUppercaseAlpha, or keyLowercaseAlpha. Not all keyboards support all modes.

GetPowerStatus	
Syntax:	GetPowerStatus( )
Parameters:	none
Returns:	Returns 1 if external power is applied, or 0 if not.
Examples:	@ret@ = GetPowerStatus( )
Notes:	Determine whether the portable device is being powered by a cradle or charge cable.

GoToPrompt	
Syntax:	GoToPrompt(<Prompt> , <Validate>)
Parameters:	
<Prompt>	The name of the prompt to move to next.
<Validate>	Whether to validate the prompt before advancing to the prompt specified.
Returns:	Nothing
Examples:	GoToPrompt (“Prompt4” , 1)
Notes:	Causes the data collection program to go to the specified prompt. If <Validate> is not zero, the data for the prompt will be saved into the elements, and they will be validated. If the validation fails, the program will not move to the new prompt. Supported for PocketPC Devices, CE Devices, and PC Client. Not supported on DOS Devices.

Message	
Syntax:	Message(<Message> , [Caption] , [Button1] , [Button2])
Parameters:	
<Message>	The message text to display
[Caption]	The window caption to display (optional)
[Button1]	Text for the first button (optional)
[Button2]	Text for the second button. The 2nd button may be omitted by using the empty string (“”) for the <Button2> text. (optional)
Returns:	Either a 1 or a 2 depending on which button is pressed (or clicked)
Examples:	@Data@ = Message("Are you sure?", "Confirm", "<Yes>", "<No>")
Notes:	Displays a message to the user. The message text is in <Message>, and the window caption is in [Caption]. You may specify text for one or two buttons using [Button1] and [Button2]. If [Caption] is not provided, a default caption will be used. If [Button1] is not provided, an OK button will be used.If [Button2] is an empty string, only <Button1> will be displayed, and it will be centered. This function returns 1 or 2 corresponding to whether button1 or button2 is pressed. On Windows platforms, this function does not return until the Message is closed. On Android, the function returns immediately.

MinimizeProgram	
Syntax:	MinimizeProgram( )
Parameters:	None
Returns:	Nothing
Examples:	MinimizeProgram( )
Notes:	Minimizes (hides) the entire client on the device. The client continues to run in the background, but is hidden. This function applies only to Windows CE or Windows Mobile devices.

Notification	
Syntax:	Notification(<Text>)
Parameters:	
<Text>	The text to be displayed in the notification.
Returns:	Nothing
Examples:	Notification(“Notification Text”)
Notes:	Briefly shows an unobtrusive popup message with the <text> specified. The popup will be shown for a few seconds and then automatically removed. On Windows devices, a small popup window will be used. On Android devices, the native Toast notification is used. The function returns immediately and does not wait for the notification to be removed. No value is returned from this function.

PlaySound	
Syntax:	PlaySound(<Soundfile>, [async])
Parameters:	
<Soundfile>	The name of a WAV file to play
[async]	Optional. Specifies whether the sound should be played synchronously (if zero - the script waits for the sound to be completed before returning) or asynchronously (if non-zero - the script continues while the sound is being played). If the parameter is not specified, the sound will be synchronous.
Returns:	Nothing
Examples:	PlaySound(“sound.wav”)
Notes:	Causes the device to play a WAV file. Supported on the Windows CE and Pocket PC devices. The WAV file must be in the program directory, or the device will simply beep.

SetBackLightMode	
Syntax:	SetBacklightMode(<mode>)
Parameters:	
<mode>	Set the backlight mode for devices that support it. If 0, the device will control the backlight according to the settings in the Control Panel or Settings applet. If 1, the backlight will stay on all the time. Stays in effect until the program is closed, or SetBackLightMode is called with a different mode.
Returns:	Nothing
Examples:	SetBackLightMode(1)
Notes:	Causes the device to set the selected backlight mode. Not all devices support controlling the backlight mode.

SetKeyboardMode	
Syntax:	SetKeyboardMode(<mode>)
Parameters:	
<mode>	The mode to set.
Returns:	Nothing
Examples:	SetKeyboardMode(keyUppercaseAlpha)
Notes:	Causes the device to set the selected keyboard mode. Not all devices support all keyboard modes.

SetPowerdownMode	
Syntax:	SetPowerDownMode(<mode>)
Parameters:	
<mode>	The powerdown mode to set.
Returns:	Nothing
Examples:	SetPowerDownMode(pwrAlwaysOn)
Notes:	Sets the powerdown mode for device which support it. The mode are pwrStandard, the device will suspend normally after inactivity. pwrUnattended, the device will turn of the screen and keyboard but stay running. pwrAlwaysOn, the device will not suspend.

SetPromptBackground	
Syntax:	SetPromptBackground(<Color>)
Parameters:	
<Color>	The color to set as the background of the prompt.
Returns:	Nothing
Examples:	SetPromptBackground(colorRed)
Notes:	Changes the background color of the current prompt. The color will be used for the background of the prompt until the program moves to a different prompt. At that point, the original colors are used again.

SetStatusBarColor	
Syntax:	SetStatusBarColor(<Color>)
Parameters:	
<Color>	The color to set as the background of the status bar.
Returns:	Nothing
Examples:	SetStatusBarColor(colorRed)
Notes:	Changes the background color of the status bar, for device which support it. The color will be used for the background of the status bar. This is supported on Android V5 and higher.

Tone	
Syntax:	Tone(<Frequency>, <Duration>, <Volume>)
Parameters:	
<Frequency>	The frequency of the tone to play.
<Duration>	The duration of the tone in milliseconds.
<Volume>	The volume to use, from 0 to 100. Not all devices support a volume adjustment.
Returns:	Nothing
Examples:	Tone(1000, 500, 100)
Notes:	Causes the device to play a tone at the specified frequency for the specified duration.

Trace	
Syntax:	Trace(<Message>)
Parameters:	
<Message>	The text to be written to the output window.
Returns:	Nothing
Examples:	Trace("Message to output")
Notes:	Displays a message in the output window in the Simulator. Has no effect on a device client.

WaitCursor	
Syntax:	WaitCursor(<set>)
Parameters:	
<set>	Whether to set or clear the wait cursor. Non-zero sets the cursor, zero removes it.
Returns:	Nothing
Examples:	WaitCursor(1)
Notes:	Used to control the Wait Cursor. This is generally used when you know the script will take a long time to complete. Be sure to remove the cursor if it is set, otherwise the cursor may be visible after your script exits.

6.10 Omni Functions

CreateValidationRemote	
Syntax:	CreateValidationRemote(<File>, <Field1>, <Value1>, ...)
Parameters:	
<File>	Validation filename to create
<Field1>	Prompt or element name of the field in the collected data file
<Value1>	Value to match on
…	You may specify up to 5 Field/Value pairs
Returns:	Returns the number of matching records
Examples:	CreateValidationRemote("val.txt", "sku", "12345") creates an updated validation file based on a defined validation file where the “sku” field is “12345”.
Notes:	Creates a validation file on the remote server and copies it to the device. The filename without path should be in <File>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen, even if it is not being used on the Advanced Prompt Settings screen.

CreateValidationRemoteBulk	
Syntax:	CreateValidationRemoteBulk( )
Parameters:	None
Returns:	Returns the number of files generated
Examples:	CreateValidationRemoteBulk( )
Notes:	Creates validation files on the remote server and copies them to the device. The filenames generated are the list setup with CreateValidationRemoteBulkAddFile.

CreateValidationRemoteBulkAddFile	
Syntax:	CreateValidationRemoteBulkAddFile(<File>, <Field1>, <Value1>, ...)
Parameters:	
<File>	Validation filename to create
<Field1>	Prompt or element name of the field in the collected data file
<Value1>	Value to match on
Returns:	Nothing
Examples:	CreateValidationRemoteBulkAddFile("val.txt", "sku", "12345")
Notes:	Adds a validation file and match fields to the list of files to be generated by CreateValidationRemoteBulk. The filename without path should be in <file>. The fields to match are given in <Field1>, <Field2>, etc. The value of each field is given in <Value1>, <Value2>, etc. Up to 5 match fields can be specified in this way. The validation file must have been defined in the Validation Files screen.

CreateValidationRemoteBulkInitialize	
Syntax:	CreateValidationRemoteBulkInitialize( )
Parameters:	None
Returns:	Nothing
Examples:	CreateValidationRemoteBulkInitialize( )
Notes:	Initializes the file list for CreateValidationRemoteBulk.

GetRemoteFailCode	
Syntax:	GetRemoteFailCode( )
Parameters:	No parameters
Returns:	Returns 1 if the last Remote function connected successfully, returns 0 if the remote connection failed while trying to execute the function.
Examples:	@ret@ = GetRemoteFailCode( )
Notes:	Retrieves the success or failure of the last remote function call that connects to the Omni Server. Functions that affect GetRemoteFailCode include: GetFileRemote, OmniSendCollectedData, RemoteGetFile, RemoteSQL, RemoteScript, RemoteScriptFile, CreateValidationFileRemote, LookupValidation and LookupValidationRecord.

GetRemoteFailCodeEx	
Syntax:	GetRemoteFailCodeEx( )
Parameters:	No parameters
Returns:	Retrieves the success or failure of the last call with connects to the Omni Server, returning a detailed error code.
Examples:	@ret@ = GetRemoteFailCodeEx( )
Notes:	Return values: 0 : No Error 1 : Could not connect to the OMNI Server 2 : OMNI Server demo limit reached 3 : No license for the Device on the OMNI Server 4 : OMNI Server not configured for the program 5 : Requested file operation is unknown 6 : Client/Server communications version mismatch 9 : Other Errors Check the OMNI Server logs for more detail if one of these errors is returned.

GetRemoteFailText	
Syntax:	GetRemoteFailText( )
Parameters:	No parameters
Returns:	Retrieves a text description for the status of the last OMNI Server communication
Examples:	@ret@ = GetRemoteFailText( )
Notes:	This text is returned in the currently selected client language.

GSMSignalStrength	
Syntax:	GSMSignalStrength( )
Parameters:	No parameters
Returns:	Returns the signal strength of the GSM/GPRS connection from 0 to 100, or -1 if not available.
Examples:	@ret@ = GSMSignalStrength( )
Notes:	Not all devices support this function.

IsAssociated	
Syntax:	IsAssociated( )
Parameters:	None
Returns:	1 if the device is associated, 0 if not, -1 if unknown or unsupported.
Examples:	@ret@ = IsAssociated( )
Notes:	Checks to see if the device is associated with an 802.11b Access Point. Check the device-specific manuals to see which devices support association checking.

IsGSMRegistered	
Syntax:	IsGSMRegistered( )
Parameters:	None
Returns:	1 if the device is registered with a GSM provider, 0 if not, -1 if unknown or unsupported.
Examples:	@ret@ = IsGSMRegistered( )
Notes:	Checks to see if the device is registered with a GSM network. This function only works on devices equipped with GSM radios and running Windows Mobile 2003 or higher with the uPhone drivers. All other devices always return -1.

OmniLoadProgram	
Syntax:	OmniLoadProgram(<progname>, <allfiles>)
Parameters:	
< progname >	The program name to load
< allfiles >	Flag indicating whether all files should be loaded.
Returns:	Returns 1 if function was successful, returns 0 if function failed
Examples:	@RetCode@ = OmniLoadProgram("Program.itb", 1)
Notes:	Connects to the Omni Server to load the program. If the device is unable to connect, an error message may be displayed depending on the setting of the Remote Failure Mode.

OmniSendCollectedData	
Syntax:	OmniSendCollectedData(<fQuiet>, [field1], [value1], ...)
Parameters:	
<fQuiet>	This parameter i no longer used.
[field1]	An optional match field to send only certain records.
[value1]	The value to match for sending only certain records.
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	OmniSendCollectedData(1)
Notes:	Forces the device to try to connect and send any collected data to the server.

OmniSendCollectedDataNoDelete	
Syntax:	OmniSendCollectedDataNoDelete(<fQuiet>, [field1], [value1], ...)
Parameters:	
<fQuiet>	This parameter is no longer used.
[field1]	An optional match field to send only certain records.
[value1]	The value to match for sending only certain records.
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	OmniSendCollectedDataNoDelete(1)
Notes:	Forces the device to try to connect and send any collected data to the server. If one or more Field/Value values are specified, only matching records will be downloaded. No records will be deleted, but the records that are sent will be marked as Sent to the Server.

OmniUpdateClient	
Syntax:	OmniUpdateClient( )
Parameters:	None
Returns:	0 if the client is already up to date. Non-zero if the client can not be updated.
Examples:	@ret@ = OmniUpdateClient( )
Notes:	Connects to the Omni server and checks for an updated Client. The client exits and restarts if an update is made, therefore the function will not return in this case.

RadioGetMode	
Syntax:	RadioGetMode( )
Parameters:	None
Returns:	Returns the currently enabled radios.
Examples:	@ret@ = RadioGetMode( )
Notes:	Returns the currently enabled radio modes on supported devices. This function only works on devices running Windows Mobile 2003 or higher. All other devices always return -1. There are Constants for the available radio modes listed in the Script Editor.

RadioSetMode	
Syntax:	RadioSetMode(<mode>)
Parameters:	
<mode>	The radio mode to set.
Returns:	Nothing
Examples:	RadioSetMode(radioWLAN)
Notes:	Enables and disables radios on supported devices. This function only works on supported devices running Windows Mobile 2003 or higher. All other devices always return -1. There are Constants for the available radio modes listed in the Script Editor.

RASConnect	
Syntax:	RASConnect(<connectionname>)
Parameters:	
<connectionname>	The name of the phonebook entry to connect, or blank to use the default connection name.
Returns:	1 if the connection is established, or 0 if not.
Examples:	@ret@ = RASConnect ("GPRS")
Notes:	Attempts a RAS connection. This can be a modem or GPRS connection. You can only have one RAS connection active at one time. If the <connectionname> is blank, the Default Connection Name specified on the device Configuration screen will be used.

RASDisconnect	
Syntax:	RASDisconnect( )
Parameters:	None
Returns:	Nothing
Examples:	RASDisconnect ( )
Notes:	Disconnects the RAS connection previously established with RASConnect. If no RAS connection is active, this function does nothing.

RASStatus	
Syntax:	RASStatus( )
Parameters:	None
Returns:	1 if the connection is established, or 0 if not.
Examples:	@ret@ = RASStatus ( )
Notes:	Checks the status of the RAS connection previously established with RASConnect. You can only have one RAS connection active at one time.

RemoteAssemblyCall	
Syntax:	RemoteAssemblyCall(<AssemblyFile>, <MethodName>, <param1>, ...)
Parameters:	
<AssemblyFile>	The Assembly DLL to load.
<MethodName>	The method to call in the Assembly.
<param1>	Optional string parameters to pass to the method.
Returns:	Returns a string result
Examples:	@Ret@ = RemoteAssemblyCall("Assembly.dll", "Class.Method", "123")
Notes:	Executes a method in an assembly located remotely on the Omni server.The Assembly DLL is located in <assemblyfile>, the method to call is in <methodname>, and the parameters are in <param1>, <param2>, etc. If the assembly name contains the full path, the OMNI Server will use it. If the assembly name does not contains a path, the OMNI Server will attempt to load it from the directory containing the ITBX. The Method name can contain a Namespace and Class name, in the form Namespace.Class.Method or Class.Method. If the method name contains a Namespace and/or Class name, the OMNI Server will search the assembly for the class and then search that class for the method. If there is no class name specified, then all classes in the assembly will be iterated to find one containing the method name. If there is more than one class containing the requested method name, it is possible that the wrong one will be called. To ensure that the right method is called, include the class name whenever possible. If the namespace is specified, the class must be specified as well. If the method is static, it will be called statically and the class will not be instantiated. If the method is not static, an instance of the class will be instantiated and the method called on it. Ensure that the assembly contains an appropriate default constructor. Returns a string containing the results of the function.

RemoteGetFile	
Syntax:	RemoteGetFile(<deviceFile>, <PCFile>)
Parameters:	
< deviceFile >	File name on the device
< PCFile >	File name on the PC with respect to the OMNI Server. If no path is specified, the file must be in the directory with the .itb file
Returns:	Returns 1 if function was successful, returns 0 if function failed
Examples:	@RetCode@ = RemoteGetFile("file123.txt", "file.txt")
Notes:	Retrieves a file from the remote OMNI Server. The file from the PC will be copied up to the device and named as specified by the <deviceFile> parameter. The file names on the PC and device need not be the same. If the remote connection is not able to be established, the function will either fail quietly or display a Retry/Cancel message depending on the behavior specified with the Remote Functions Default Failure Mode or a call to the SetRemoteFailMode function.

RemoteGetFileBulk	
Syntax:	RemoteGetFile(<FileList>)
Parameters:	
<FileList>	the name of a list containing the files to get from the OMNI Server.
Returns:	Returns 1 if function was successful, returns 0 if function failed
Examples:	RemoteGetFileBulk("FileListName")
Notes:	Retrieves a set of files from the remote OMNI Host. The list of files should be in a List named <filelist> where the key is the PC File, and the Value is the Device File. The Device Filenames should not contain a full PC path. If a path is specified, it will be removed. Regardless, the retrieved files will be stored in the ITBX folder on the Device. The PC Filenames can contain a full PC path. If a path is specified, then the server will look for the files at that path. Otherwise, the files must be in the same directory on the server as the ITBX.

RemoteGetITBVersion	
Syntax:	RemoteGetITBVersion(<program>)
Parameters:	
< program >	The file name of the ITB program on the OMNI Server
Returns:	The version string of the program.
Examples:	@ver@ = RemoteGetITBVersion("sample.itb")
Notes:	Connects to the OMNI Server and returns the Version string of the ITB on the server. This is the version string set on the Program Settings screen.

RemoteGetOmniServerVersion	
Syntax:	RemoteGetOmniServerVersion( )
Parameters:	None
Returns:	The version string of the program.
Examples:	@ver@ = RemoteGetOmniServerVersion( )
Notes:	Connects to the OMNI Server and returns the Version Number of the server. This can be used to determine if a new version has been installed, and if the client should be updated.

RemoteGetProgramList	
Syntax:	RemoteGetProgramList( )
Parameters:	None
Returns:	The list of programs on the server, delimited by TAB characters.
Examples:	@List@ = RemoteGetProgramList( )
Notes:	Connects to the OMNI Server and returns the list of programs the server is configured to use. The file names are separated by a TAB character. Only the programs that are available to this device (If device limits are used) will be returned.

RemotePutFile	
Syntax:	RemotePutFile(<deviceFile>, <PCFile>)
Parameters:	
< deviceFile >	File name on the device
< PCFile >	File name on the PC with respect to the OMNI Server. If no path is specified, the file must be in the directory with the .itb file
Returns:	Returns 1 if function was successful, returns 0 if function failed
Examples:	@RetCode@ = RemotePutFile("file123.txt", "file.txt")
Notes:	Sends a file to the remote OMNI Server. The file names on the PC and device need not be the same. If the remote connection is not able to be established, the function will either fail quietly or display a Retry/Cancel message depending on the behavior specified with the Remote Functions Default Failure Mode or a call to the SetRemoteFailMode function.

RemoteScript	
Syntax:	RemoteScript(<Function>, <Param1> ...)
Parameters:	
<Function>	Name of the function to call
<Param1>	Parameters to pass to the function
…	Additional parameters are separated by commas. The number of parameters is not limited.
Returns:	Returns a string containing the results of the function
Examples:	@Ret@ = RemoteScript("FunctionName", "ABC", "123")
Notes:	Executes a VBScript remotely on the Omni server. The function to execute is in <Function>, and the parameters are in <Param1>, <Param2>, etc. The remote function to execute must be defined in the .itb file in the Remote Script screen. If the remote connection is not able to be established, the function will either fail quietly or display a Retry/Cancel message depending on the behavior specified with the Remote Functions Default Failure Mode or a call to the SetRemoteFailMode function.

RemoteScriptFile	
Syntax:	RemoteScriptFile(<Scriptfile>, <Function>, <Param1> ...)
Parameters:	
<Scriptfile>	File located with the .itb file
<Function >	Name of the function to call
<Param1>	Parameters to pass to the function
…	Additional parameters are separated by commas. The number of parameters is not limited.
Returns:	Returns a string containing the results of the function
Examples:	@Ret@ = RemoteScriptFile("ScriptFile.txt", "FunctionName", "ABC")
Notes:	Executes a VBScript remotely on the Omni server. The function to execute is in the external file named <Scriptfile>. The function name is in <Function>, and the parameters are in <Param1>, <Param2>, etc. The script file must exist in the directory containing the .itb file. If the remote connection is not able to be established, the function will either fail quietly or display a Retry/Cancel message depending on the behavior specified with the Remote Functions Default Failure Mode or a call to the SetRemoteFailMode function.

RemoteScriptReturnFile	
Syntax:	RemoteScriptReturnFile(<devicefile>, <Function>, <Param1> ...)
Parameters:	
<devicefile>	The name of the file to be returned.
<Function >	Name of the function to call
<Param1>	Parameters to pass to the function
…	Additional parameters are separated by commas. The number of parameters is not limited.
Returns:	Returns a string containing the results of the function
Examples:	@Ret@ = RemoteScriptReturnFile("Result.txt", "FunctionName", "ABC")
Notes:	Executes a VBScript remotely on the Omni server. The function name is in <Function>, and the parameters are in <Param1>, <Param2>, etc. The script file must exist in the directory containing the .itb file. If the script runs, it returns the file name in <devicefile>. If the remote connection is not able to be established, the function will either fail quietly or display a Retry/Cancel message depending on the behavior specified with the Remote Functions Default Failure Mode or a call to the SetRemoteFailMode function.

RemoteSetClock	
Syntax:	RemoteSetClock( )
Parameters:	none
Returns:	Returns 1 if the clock was set, or 0 if not.
Examples:	@ret@ = RemoteSetClock( )
Notes:	Connects to the OMNI Server to retrieve the server date and time. If the connection succeeds, the device time is set to match.

RemoteSQL	
Syntax:	RemoteSQL(<ConnectionString>, <SqlStatement>)
Parameters:	
<ConnectionString>	Connection string to connect to the database
<SqlStatement>	SQL to execute
Returns:	Returns the first record of the result of the SQL statement. If multiple fields are returned, the fields are concatenated together with TAB characters between them. Use Split or SplitN to separate the result fields. If no records are returned by the query, an empty string will be returned.
Examples:	@Ret@ = RemoteSQL("DSN=TestDSN", "Select top 1 * from Table")
Notes:	Executes a SQL statement remotely on the Omni server. If the remote connection is not able to be established, the function will either fail quietly or display a Retry/Cancel message depending on the behavior specified with the Remote Functions Default Failure Mode or a call to the SetRemoteFailMode function.

SetServerAddress	
Syntax:	SetServerAddress(<Address>, [Permanent])
Parameters:	
<Address>	The IP Address or URL of the OMNI Server.
[Permanent]	Whether to save this address permanently.
Returns:	Nothing
Examples:	SetServerAddress("servername")
Notes:	Sets the server address to use for Omni connections. If [permanent] is zero or not specified, this address only applies while the data collection program is running. If [permanent] is non-zero, this address is saved in the configuration as the default address. If <address> is blank, the address will be reset to the default stored in the configuration. In this case, the [permanent] parameter is ignored.

WifiSignalStrength	
Syntax:	WifiSignalStrength( )
Parameters:	No parameters
Returns:	Returns the signal strength of the Wifi connection from 0 to 100, or -1 if not available.
Examples:	@ret@ = WifiSignalStrength( )
Notes:	Not all devices support this function.

6.11 Other Functions

AssemblyCall	
Syntax:	AssemblyCall(<classname>, <methodname>, <parameter>, <resultvariable>)
Parameters:	
<classname>	The namespace and class of an assembly loaded with AssemblyLoad.
<methodname>	The name of the method to call.
<parameter>	The parameter to pass to the assembly method.
<resultvariable>	A variable to receive the result data.
Returns:	The Int32 result of the method call.
Examples:	@ret@ = AssemblyCall("Namespace.Class", "Method", "ABCDEFG", @Result@)
Notes:	Calls a method on a class loaded using AssemblyLoad. The <classname> parameter must specify the namespace and class to be created in the form "Namespace.Class". The method prototype must be Int32 Method(string Parameter, out string Result)

AssemblyLoad	
Syntax:	AssemblyLoad(<AssemblyFile>, <classname>)
Parameters:	
<AssemblyFile>	The name ITB program to call.
<classname>	The name of the class to load from the assembly.
Returns:	1 if the DLL was loaded, or 0 if not.
Examples:	AssemblyLoad("Assembly.dll", "Namespace.Class")
Notes:	Load an external Assembly DLL and class. This library must be located in the ITB directory. The <classname> parameter must specify the namespace and class to be created in the form "Namespace.Class".

CallITB	
Syntax:	CallITB(<itbfile>)
Parameters:	
<itbfile>	The name ITB program to call.
Returns:	1 if the ITB was loaded, or 0 if not.
Examples:	CallITB ("Secondary.itb")
Notes:	This function loads a second ITB and starts data collection for it. The new ITB will not share any data or variables with the first one. This function does not return until the second ITB exits.

DLLCall	
Syntax:	DLLCall(<dllfile>, <functionname>, <inparam>, <outparam>)
Parameters:	
<dllfile>	The name of the DLL containing the function to call.
<functionname>	The name of the DLL function to call.
<inparam>	A string containing the parameters to pass to the function.
<outparam>	A variable to receive the returned result.
Returns:	The return code of the function.
Examples:	DLLCall ("Custom.dll", "Function1", "This data goes to the DLL", @ReturnedData@)
Notes:	The prototype for the DLL function must be: _stdcall DWORD DLLFunction(const WCHAR *wszInParam, WCHAR *wszOutParam); If the DLL was not already loaded using DLLLoad, it will be loaded, the function called, then released. If you need to call a DLL Function multiple times, you will get better performance by calling DLLLoad first.

DLLLoad	
Syntax:	DLLLoad(<dllfile>)
Parameters:	
<dllfile>	The DLL File that is to be loaded.
Returns:	Returns 1 if the DLL was loaded, or 0 if there was an error.
Examples:	DLLLoad ("Custom.dll")
Notes:	The DLL must be located in the ITB directory. The DLL is released by calling DLLRelease. The DLL is also released automatically when the data collection program exits.

DLLRelease	
Syntax:	DLLRelease(<dllfile>)
Parameters:	
<dllfile>	The filename of a previously loaded DLL.
Returns:	Nothing
Examples:	DLLRelease ("custom.dll")
Notes:	Releases a DLL loaded using DLLLoad.

Exec	
Syntax:	Exec(<text>)
Parameters:	
<text>	Script code to execute
Returns:	Nothing
Examples:	Exec("@Test@ = Left("ABCDE", 2)")
Notes:	Executes a single line of script code.

GetExternalITBVersion	
Syntax:	GetExternalITBVersion(<filename>)
Parameters:	
<filename>	The name of the ITB program to check.
Returns:	The version string of the program.
Examples:	@ver@ = GetExternalITBVersion("sample.itb")
Notes:	Returns the version string set on the Program Settings screen.

GetLanguage	
Syntax:	GetLanguage( )
Parameters:	None
Returns:	Returns the currently selected language used by the program. The language must have been set with SetLanguage.
Examples:	@ret@ = GetLanguage( )
Notes:	The default language name is "Default".

GetStringTableEntry	
Syntax:	GetStringTableEntry(<key>, [languagename])
Parameters:	
<key>	The name of the string to search for.
[languagename]	Optional. The name of the string table to search.
Returns:	Returns the string located.
Examples:	@ret@ = GetStringTableEntry("prompt.element.text")
Notes:	Gets a string from the string table. The string to retrieve is given by <key>. If the [languagename] parameter is not specified, the currently selected language chosen by SetLanguage will be used. If no entry has been specified for the key in that language, the default language will be used instead. If the [languagename] parameter is specified, the string for that language will be returned. If no entry has been specified for the key in that language, an empty string will be returned. If the [languagename] is invalid, a blank will be returned. The [languagename] is case-sensitive. The default language name is "Default".

GlobalScript	
Syntax:	GlobalScript(<scriptname>)
Parameters:	
<scriptname>	The name of the global script to execute.
Returns:	Returns 1 if the global script was executed, or 0 if it could not be found
Examples:	GlobalScript("TestScript")
Notes:	Executes a Global Script. The <scriptname> parameter specifies a script defined on the Global Scripts screen.

GlobalScriptFile	
Syntax:	GlobalScriptFile(<scriptname>, <filename>)
Parameters:	
<scriptname>	The name of the global script to execute.
<filename>	The name of the external file containing the global scripts.
Returns:	Returns 1 if the global script was executed, or 0 if it could not be found
Examples:	GlobalScriptFile("TestScript", “Script.txt”)
Notes:	Executes a Global Script from an external file. The <scriptname> parameter specifies the name of the script, and the <filename> is the name of the external file. The file must be located in the same directory as the ITB file.

GlobalTimerInterval	
Syntax:	GlobalTimerInterval(<interval>)
Parameters:	
<interval>	The interval, in milliseconds, to use for the Global Timer event.
Returns:	Nothing
Examples:	GlobalTimerInterval(30000)
Notes:	Changes the Interval used by the Global Timer event. Setting the Interval to 0 disables the Global Timer. The Interval is in milliseconds. When the Interval is set, the currently waiting timer is stopped, and a new timer is started. This means that the next execution of the Global Timer will be when the interval elapses from the time it was set.

Guid	
Syntax:	Guid([brackets])
Parameters:	
[brackets]	Optional. Whether the Guid should be wrapped in brackets.
Returns:	The Guid string
Examples:	@guid@ = Guid( )
Notes:	Creates a new GUID (Globally Unique Identifier). A GUID is a unique 128-bit number that can be used as a reference number. If the [bracket] parameter is non-zero, the GUID format will be {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXX}. If the [bracket] parameter is zero or not provided, the GUID format will be XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXX.

ListAdd	
Syntax:	ListAdd(<listname>, <key>, <value>)
Parameters:	
<listname>	The name of the list to add the key/value pair.
<key>	The Key to add to the list.
<value>	The value to associate with the key.
Returns:	Nothing
Examples:	ListAdd("List", "Key", "Value")
Notes:	Adds a key/value pair to a list. If the key does not already exist in the list, it is added. If the key already is in the list, the value is updated to this new value.

ListClear	
Syntax:	ListClear(<listname>)
Parameters:	
<listname>	The name of the list to clear.
Returns:	Nothing
Examples:	ListClear("List")
Notes:	Removes all key/value pairs from the list.

ListCount	
Syntax:	ListCount(<listname>)
Parameters:	
<listname>	The name of the list to clear.
Returns:	The number of items in the list
Examples:	@Count@ = ListCount("List")
Notes:	Returns the number of items in the list. Returns zero if the list is not found, or no items are found.

ListGetKeyAt	
Syntax:	ListGetKeyAt(<listname>, <index>)
Parameters:	
<listname>	The name of the list to lookup from.
<index>	The index of the key to retreive.
Returns:	The key at the specified index.
Examples:	@ret@ = ListGetKeyAt("List", "2")
Notes:	Get the key for a specific index position in the list named by <listname>. This function can be used to iterate the keys in a list. The keys are not sorted. If the <index> is out of range, an empty string is returned.

ListLookup	
Syntax:	ListLookup(<listname>, <key>)
Parameters:	
<listname>	The name of the list to lookup from.
<key>	The Key to lookup.
Returns:	The value found in the list.
Examples:	@ret@ = ListLookup("List", "Key")
Notes:	Looks up a value from the list named in <listname>. The value to lookup is named in <key>. If the value exists, it is returned. If not, an empty string is returned.

OpenMapToAddress	
Syntax:	OpenMapToAddress(<address>)
Parameters:	
<address>	The address to find.
Returns:	Nothing
Examples:	OpenMapToAddress("123 Main St, City State, zipcode")
Notes:	Open a map to an address. Uses Google Maps on most platforms.

OpenURLInBrowser	
Syntax:	OpenURLInBrowser(<url>, [separatebrowser])
Parameters:	
<url>	The URL to open
[separatebrowser]	Optional. If 1, open is a separate browser. Otherwise open within the client.
Returns:	Nothing
Examples:	OpenURLInBrowser(http://www.example.com)
Notes:	Open the <url> in the default web browser. The optional [separatebrowser] parameter applies to Android only. If 1, the browser will be launched as a separate task. Otherwise, it will be launched within the client. The default is to launch the browser within the client.

Ping	
Syntax:	Ping(<address>)
Parameters:	
<address>	The address to ping.
Returns:	Returns 1 if the ping succeeds, or 0 if the ping fails.
Examples:	Ping("192.168.1.200")
Notes:	Attempts to ping the IP Address or Computer Name specified by <address>. Ping only works over an established network connection, and does not work over Bluetooth.

RAMFreeSpace	
Syntax:	RAMFreeSpace( )
Parameters:	None
Returns:	The free RAM reported by the operating system.
Examples:	@ret@ = RAMFreeSpace( )
Notes:	Returns the amount of free RAM.

RAMTotalSize	
Syntax:	RAMTotalSize( )
Parameters:	None
Returns:	The total RAM size reported by the operating system.
Examples:	@ret@ = RAMTotalSize( )
Notes:	Returns the total size of RAM.

SendEmailDefault	
Syntax:	SendEmailDefault(<toaddress>, <ccaddress>, <subject>, <body>, [attachment])
Parameters:	
<toaddress>	The address to send the mail to.
<ccaddress>	The optional CC address. Leave blank if you do not want to CC the message.
<subject>	the message subject.
<body>	The message body.
[attachment]	Optional. The filename to be attached.
Returns:	Nothing
Examples:	SendEmailDefault("to@example.com", "", Email Subject, Email Body)
Notes:	Opens a new email in the system's default Email client. The <toaddress>, <subject> and <body> are required. The <ccaddress> can be left blank. The [attachment] allows you to attach a file to the email, if the system permits it. This function is intended for use where a short email is required, and the user has a chance to edit the message before sending. Files can be attached in Android, but not on the PC, Windows CE or Window Mobile. If file attachments are required on these systems, use SendEmailDirect.

SendEmailDirect	
Syntax:	SendEmailDirect(<toaddress>, <ccaddress>, <fromaddress>, <serveraddress>, <serverport>, <subject>, <body>, [bodyisfile], [attachment], [username], [password], [domain], [usessl])
Parameters:	
<toaddress>	The address to send the mail to.
<ccaddress>	The optional CC address. Leave blank if you do not want to CC the message.
<fromaddress>	The email address to send From.
<serveraddress>	The SMTP Server address
<serverport>	The SMTP Server port.
<subject>	The message subject.
<body>	The message body.
[bodyisfile]	Optional, If 1, the file specified in [attachment] will be loaded as the message body.
[attachment]	Optional. The filename to be attached.
[username]	Optional. The mail server username.
[password]	Optional. The mail server password.
[domain]	Optional. The main server domain.
[usessl]	Optional. Set to 1 if using an SSL connection.
Returns:	Nothing
Examples:	SendEmailDirect("to@example.com", "", "from@example.com", "mail.example.com", "25", Email Subject, Email Body)
Notes:	Attempts to directly send an email through a mail server without any user intervention. The <toaddress>, <fromaddress>, <serveraddress>, <serverport>, <subject> and <body> are required. The <ccaddress> can be left blank. If the [bodyisfile] is non-zero, the contents of the body field are assumed to be a filename and the contents of that file are loaded and used as the HTML body of the email. The [attachment] allows you to attach a file to the email. The [username], [password], and [domain], and [usessl] parameters are required if your SMTP server uses authentication. On Windows CE/Windows Mobile, the Domain is required. Attachments are not supported on Windows CE/Windows Mobile. SSL connections are not supported on Windows CE/Windows Mobile.

SetAlias	
Syntax:	SetAlias(<Alias>, [Permanent])
Parameters:	
<Alias>	The new alias to use for the device.
[Permanent]	Optional. Whether to make the alias change permanent.
Returns:	Nothing
Examples:	SetAlias("Device1")
Notes:	Sets the Alias to use for this device when collecting data. If [permanent] is zero or not specified, this alias only applies while the data collection program is running. If [permanent] is non-zero, this alias is saved in the configuration as the default alias. If <alias> is blank, the alias will be reset to the default stored in the configuration. In this case, the [permanent] parameter is ignored.

SetLanguage	
Syntax:	SetLanguage([LanguageName])
Parameters:	
[LanguageName]	Optional. The string table to select.
Returns:	Data in the itscript.ret file, if waiting for a response.
Examples:	SetLanguage("English")
Notes:	Changes the language used by the program. This controls the string table column that will be used for displayed text for elements, and by the GetString function. If [languagename] is not specified, or does not exist in the string table, the default language will be selected. The [languagename] is case-sensitive. The default language name is "Default".

Shell	
Syntax:	Shell(<CommandLine>, [NoWait])
Parameters:	
<CommandLine>	Command line for external program to run
[NoWait]	Optional parameter. Pass 1 if you do not want to wait for a result. 0 or no parameter will wait for the result.
Returns:	Data in the itscript.ret file, if waiting for a response.
Examples:	Shell("MyProgram.exe")
Notes:	Shells to an external program. You may specify command line and parameters. The external program should write the data to be returned in a file named "itscript.ret". The Shell function will read that data and return it.

ShowSIP	
Syntax:	ShowSIP(<show>)
Parameters:	
<show>	Specifies whether to show or hide the Soft Input Panel.
Returns:	Nothing
Examples:	ShowSIP(1)
Notes:	Show or hide the Pocket PC Soft Input Panel. The <show> parameter specifies whether to show or hide the panel. Specify 1 or TRUE to show the panel, and 0 or FALSE to hide it. This function only works for devices that support the Soft Input Panel.

Sleep	
Syntax:	Sleep(<delay>)
Parameters:	
<delay>	The number of milliseconds to delay.
Returns:	Nothing
Examples:	Sleep(250)
Notes:	Sleeps the script for the specified number of milliseconds. This function should generally be used only in a Background element Processing script.

SocketClose	
Syntax:	SocketClose( )
Parameters:	None
Returns:	Nothing
Examples:	SocketClose( )
Notes:	Closes the socket opened by SocketOpen.

SocketOpen	
Syntax:	SocketOpen(<ipaddress>, <ipport>)
Parameters:	
<ipaddress>	The ipaddress to attempt to connect to.
<ipport>	the ipport to attempt to connect to.
Returns:	Returns 1 if the socket was opened, or 0 if not.
Examples:	@ret@ = SocketOpen("127.0.0.1", "9100")
Notes:	Attempts to open a socket the specified <ipaddress> and <ipport>. The <ipaddress> can be either an IP Address or a machine name. The system supports one open socket at a time. If a socket is already open when SocketOpen is called, the previous socket is closed and the new socket opened.

SocketRead	
Syntax:	SocketRead(<variable>)
Parameters:	
<variable>	The variable to receive the data read from the socket.
Returns:	Returns the number of bytes read from the socket. Returns 0 if no data was available, or -1 if there was an error.
Examples:	@ret@ = SocketRead(@result@)
Notes:	Attempts to read from the socket opened by SocketOpen. The data will be placed into the variable named in <variable>.

SocketWrite	
Syntax:	SocketWrite(<data>)
Parameters:	
<data>	The data to write to the socket.
Returns:	Returns the number of bytes written to the socket.
Examples:	@ret@ = SocketClose( )
Notes:	Writes the data in the string <data> to the socket opened by SocketOpen.

StorageFreeSpace	
Syntax:	StorageFreeSpace( )
Parameters:	none
Returns:	The number of bytes of free space remaining in the file store.
Examples:	@ret@ = StorageFreeSpace( )
Notes:	Returns the amount of free space (in bytes) of the file store containing the ITB program.

StorageTotalSpace	
Syntax:	StorageTotalSpace( )
Parameters:	none
Returns:	The total size of the file store.
Examples:	@ret@ = StorageTotalSpace( )
Notes:	Returns the total amount of space (in bytes) of the file store containing the ITB program.

TableAddField	
Syntax:	TableAddField(<tablename>, <fieldname>)
Parameters:	
<tablename>	The name of the table to add to.
<fieldname>	The field name to add.
Returns:	Nothing
Examples:	TableAddField("TableName", "FieldName")
Notes:	Adds a field to a Table. The name of the field to add is in <fieldname> and the table name is <tablename>. If the table does not exist, it will be created. If the <fieldname> already exists in the table, nothing happens. Does not return a value.

TableAddRow	
Syntax:	TableAddRow(<tablename>)
Parameters:	
<tablename>	The name of the table to add the row to.
Returns:	Returns the index of the new row. Indexes in ITScriptNet are 1-based, so the first row in the table is row 1 (not 0).
Examples:	TableAddRow("TableName")
Notes:	Adds a new blank row to a Table. The table name is in <tablename>. If the table does not exist, it will be created and will have no fields.

TableClone	
Syntax:	TableClone(<sourcetablename>, <newtablename>)
Parameters:	
<sourcetablename>	The name of the table to use as the source.
<newtablename>	The name of the new table to create.
Returns:	Nothing
Examples:	TableClone("SourceTableName", "NewTableName")
Notes:	Creates a new Table with the same structure as an existing table. The source table name is in <sourcetablename>. The new table name is in <newtablename>. The new table will have exactly the same fields as the source table, but no rows. You can add fields with TableAddField, and add rows with TableAddRow.

TableCopyRow	
Syntax:	TableCopyRow(<sourcetablename>, <sourcerow>, <destinationtablename>)
Parameters:	
<sourcetablename>	The name of the table to use as the source.
<sourcerow>	The index of the source row.
<destinationtablename>	The name of the destination table to copy the row into.
Returns:	Nothing
Examples:	TableCopyRow("SourceTableName", "4", "NewTableName")
Notes:	Copies the row from the table named <sourcetablename> at index <sourcerow> to the table named <destinationtablename>. The row is appended to the end of the destination table. Field data is copied by field name, so the fields in the source and destination tables should match. Any fields in the destination table that are not in the source table will be left blank. Indexes in ITScriptNet are 1-based, so the first row in the table is row 1 (not 0).

TableCreate	
Syntax:	TableCreate(<tablename>)
Parameters:	
<tablename>	The name of the table to create.
Returns:	Nothing
Examples:	TableCreate("TableName")
Notes:	Creates an empty Table. The table name is in <tablename>. The table will have no fields and no rows. You can add fields with TableAddField, and add rows with TableAddRow.

TableDelete	
Syntax:	TableDelete(<tablename>)
Parameters:	
<tablename>	The name of the table to delete.
Returns:	Nothing
Examples:	TableDelete("TableName")
Notes:	Removes the table <tablename> from memory. This completely deletes the table, and all rows and fields. If the table does not exist, nothing happens.

TableFindRow	
Syntax:	TableFindRow(<tablename>, <fieldname>, <value>)
Parameters:	
<tablename>	The name of the table to search.
<fieldname>	The field to search.
<value>	The value to search for.
Returns:	The index of the row containing the value.
Examples:	@ret@ = TableFindRow("TableName", "FieldName", "Value")
Notes:	Searches a table for the first row with the value <value> in the field <fieldname>. The table name is in <tablename>. Returns the index of the row. Indexes in ITScriptNet are 1-based, so the first row in the table is row 1 (not 0).

TableGetField	
Syntax:	TableGetField(<tablename>, <row>, <field>)
Parameters:	
<tablename>	The name of the table to search.
<row>	The row to retrieve.
<field>	The field to return the value of.
Returns:	The value of the specified field in the specified row.
Examples:	@ret@ = TableGetField("TableName", "2", "FieldName")
Notes:	Gets the value of the field <fieldname> in row number <row>. The table name is in <tablename>. Indexes in ITScriptNet are 1-based, so the first row in the table is row 1 (not 0).

TableRemoveAllRows	
Syntax:	TableRemoveAllRows(<tablename>)
Parameters:	
<tablename>	The name of the table to clear.
Returns:	Nothing
Examples:	TableRemoveAllRows("TableName")
Notes:	Removes all rows from the table <tablename>. The fields are retained, but all rows are removed. If the table does not exist, nothing happens.

TableRemoveField	
Syntax:	TableRemoveField(<tablename>, <field>)
Parameters:	
<tablename>	The name of the table to search.
<field>	The name of the field to remove.
Returns:	Nothing
Examples:	TableRemoveField("TableName", "Field")
Notes:	Removes the field named <field> from the table <tablename>.

TableRemoveRow	
Syntax:	TableRemoveRow(<tablename>, <row>)
Parameters:	
<tablename>	The name of the table to search.
<row>	The index of the row to remove.
Returns:	Nothing
Examples:	TableRemoveRow("TableName", 3)
Notes:	Removes the row <row> from the table <tablename>. Indexes in ITScriptNet are 1-based, so the first row in the table is row 1 (not 0).

TableRowCount	
Syntax:	TableRowCount(<tablename>)
Parameters:	
<tablename>	The name of the table to check.
Returns:	The number of rows in the table.
Examples:	@ret@ = TableRowCount("TableName")
Notes:	Returns the number of rows in the table named <tablename>.

TableRowToList	
Syntax:	TableRowToList(<tablename>, <row>, <listname>)
Parameters:	
<tablename>	The name of the table to check.
<row>	The row to copy to the list
<listname>	The name of the list to copy the data into.
Returns:	Nothing
Examples:	TableRowToList("TableName", "2", "ListName")
Notes:	Retrieves the row <row> from the table <tablename> and stores it in a List named <listname>. The Keys for the list are the field names of the table, and the Values are the row field values. Indexes in ITScriptNet are 1-based, so the first row in the table is row 1 (not 0).

TableSetField	
Syntax:	TableSetField(<tablename>, <row>, <fieldname>, <value>)
Parameters:	
<tablename>	The name of the table to check.
<row>	The row to copy to the list.
<fieldname>	The name of the field to set.
<value>	The value to put in the field.
Returns:	Nothing
Examples:	TableSetField("TableName", "2", "FieldName", "Value")
Notes:	Sets the value of the field <fieldname> in row <row> in the table <tablename>.

VoiceCall	
Syntax:	VoiceCall(<phonenumber>)
Parameters:	
<phonenumber>	The phone number to call.
Returns:	Nothing
Examples:	VoiceCall("555-555-5555")
Notes:	Place a voice call. Supported on Windows Mobile and Android clients only.

6.12 Print Functions

AirPrintFile	
Syntax:	AirPrintFile(<filename>)
Parameters:	
<filename>	The name of the file to print using AirPrint (iOS Only).
Returns:	1 if the print was successful, or 0 if not.
Examples:	AirPrintFile("file.pdf")
Notes:	Sends the contents of the disk file named <filename> to an AirPrint printer (iOS Only). Returns 1 if the print was successful, or 0 if not.

BTGetDeviceList	
Syntax:	BTGetDeviceList(<listname>, [rundiscovery])
Parameters:	
<listname>	The name of the list to receive the printers.
[rundiscovery]	Set to 1 to run discovery.
Returns:	Nothing
Examples:	BTGetDeviceList("DeviceList")
Notes:	Retrieves the list of paired Bluetooth devices, placing them into the list named <listname>. The list is cleared before any items are added. The list is keyed by the Bluetooth MAC address in the form "00:00:00:00:00:00", and the value is the Bluetooth name of the device. If the optional [rundiscovery] parameter is non-zero, runs Discovery and returns both the paired and discovered devices. The Discovery process takes several seconds to complete, during which the script will block. Note: This parameter has no effect on Android, which always returns the Paired (Bonded) device list.

BTPrtFile	
Syntax:	BTPrtFile(<Printername>, <Filename>, [evaluate])
Parameters:	
<Printername>	The name or MAC Address of the Bluetooth Printer.
<Filename>	File on device to send to IrDA port, typically to a printer.
[evaluate]	Set to 1 to substitute variables in the file data.
Returns:	Returns 1 if the print was successful, returns 0 if print failed
Examples:	BTPrtFile("PrinterName", "File.prn", 1)
Notes:	Sends the contents of the disk file named <filename> to a Bluetooth. If the optional [evaluate] parameter is non-zero, performs variable substitution before sending. Otherwise no translations or substitutions are performed. The <printername> parameter can specify either the printer's Bluetooth name when paired, or the printer's Bluetooth MAC address in the form "00:00:00:00:00:00".

BTPrtPrint	
Syntax:	BTPrtPrint(<Printername>, <Printfile>)
Parameters:	
<Printername>	The name or MAC Address of the Bluetooth Printer.
<Printfile>	File containing printer-specific commands, typically generated in a Label Design package. The PrintFile is defined in the Print Files screen.
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	BTPrtPrint("PrinterName", "PrintFile")
Notes:	Sends <printfile>, defined in the Print Files screen, to a Bluetooth printer. Performs variable substitution before sending. The <printername> parameter can specify either the printer's Bluetooth name when paired, or the printer's Bluetooth MAC address in the form "00:00:00:00:00:00".

BTPrtString	
Syntax:	BTPrtString(<Printername>, <String>)
Parameters:	
<Printername>	The name or MAC Address of the Bluetooth Printer.
<String>	String to send to the Bluetooth printer.
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	BTPrtString("PrinterName", "ABCDEFG")
Notes:	Sends <string> to a Bluetooth printer. The <printername> parameter can specify either the printer's Bluetooth name when paired, or the printer's Bluetooth MAC address in the form "00:00:00:00:00:00".

RFPrtFile	
Syntax:	RFPrtFile(<Address>, <Port>, <Filename>)
Parameters:	
<Address>	Printer IP Address
<Port>	Port to use
<Filename>	File on device to send
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	RFPrtFile("192.168.1.1", "6101", "File.prn")
Notes:	Sends the contents of the disk file named <Filename> to an RF printer. No translations or substitutions are performed. The printer IP Address should be in <Address> and the port in <Port>. If the RF connection is not able to be made, the Retry/Cancel message will be displayed to the user. The printing functions do not do anything in the simulator.

RFPrtPrint	
Syntax:	RFPrtPrint(<Address>, <Port>, <PrintFile>)
Parameters:	
<Address>	Printer IP Address
<Port>	Port to use
<PrintFile>	File containing printer-specific commands, typically generated in a Label Design package. The printfile is defined in the Print Files screen.
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	RFPrtPrint("192.168.1.1", "6101", "PrintFile")
Notes:	Sends <PrintFile>, defined in the Print Files screen, to an RF printer. Performs variable substitution before sending. The printer IP Address should be in <Address> and the port in <Port>. If the RF connection is not able to be made, the Retry/Cancel message will be displayed to the user. The printing functions do not do anything in the simulator.

RFPrtString	
Syntax:	RFPrtString(<Address>, <Port>, <String>)
Parameters:	
<Address>	Printer IP Address
<Port>	Port to use
<String>	String to send
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	RFPrtString("192.168.1.1", "6101", "ABCDEFG")
Notes:	Sends <String> to an RF printer. The printer IP Address should be in <Address> and the port in <Port>. If the RF connection is not able to be made, the Retry/Cancel message will be displayed to the user. The printing functions do not do anything in the simulator.

SerialPrtFile	
Syntax:	SerialPrtFile(<Port>, <Baud>, <Parity>, <Bits>, <Filename>)
Parameters:	
<Port>	Number indicating the serial port to print to
<Baud>	Baud rate
<Parity>	Parity setting
<Bits>	Number of data bits
<Filename>	File on device to send to serial port, typically to a printer
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	SerialPrtFile(serCOM1, SerBaud9600, serParityNone, serData8, "File.prn")
Notes:	Sends the contents of the disk file named <Filename> to a Serial printer. No translations or substitutions are performed. The serial port is in <Port>, the baud rate in <Baud>, the parity in <Parity> and data bits in <Bits>. If the serial connection is not able to be made, the Retry/Cancel message will be displayed to the user.

SerialPrtPrint	
Syntax:	SerialPrtPrint(<Port>, <Baud>, <parity>, <Bits>, <PrintFile>)
Parameters:	
<Port>	Number indicating the serial port to print to
<Baud>	Baud rate
<Parity>	Parity setting
<Bits>	Number of data bits
<PrintFile>	File containing printer-specific commands, typically generated in a Label Design package. The PrintFile is defined in the Print Files screen.
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	SerialPrtPrint(serCOM1, SerBaud9600, serParityNone, serData8, "PrintFile")
Notes:	Sends <Printfile>, defined in the Print Files screen, to a Serial printer. Performs variable substitution before sending. The serial port is in <Port>, the baud rate in <Baud>, the parity in <Parity> and data bits in <Bits>. If the serial connection is not able to be made, then a Retry/Cancel message will be displayed to the user.

SerialPrtString	
Syntax:	SerialPrtString(<Port>, <Baud>, <Parity>, <Bits>, <String>)
Parameters:	
<Port>	Number indicating the serial port to print to
<Baud>	Baud rate
<Parity>	Parity setting
<Bits>	Number of data bits
<String>	String to send to the serial port (typically to a printer)
Returns:	Returns 1 if the function was successful, returns 0 if function failed
Examples:	SerialPrtString(serCOM1, serBaud9600,serParityNone, serData8,"ABCDEFG")
Notes:	Sends <String> to a Serial printer. The serial port is in <Port>, the baud rate in <Baud>, the parity in <Parity> and data bits in <Bits>. If the serial connection is not able to be made, then a Retry/Cancel message will be displayed to the user.

6.13 Report Functions

ReportConvertFileToHTML	
Syntax:	ReportConvertFileToHTML(<reportfile>, <outputfile>, [dpu])
Parameters:	
<reportfile>	The file name of the report to convert.
<outputfile>	The output file to write.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToHTML(ReportFile.rptx, OutFile.html)
Notes:	Converts the external Report file named <reportfile> to HTML. The result will be written to the file specified in <outputfile>. The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertFileToPCL	
Syntax:	ReportConvertFileToPCL(<reportfile>, <outputfile>, [dpu])
Parameters:	
<reportfile>	The file name of the report to convert.
<outputfile>	The output file to write.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPCL(ReportFile.rptx, OutFile.PCL)
Notes:	Converts the external Report file named <reportfile> to PCL. The result will be written to the file specified in <outputfile>. The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertFileToPDF	
Syntax:	ReportConvertFileToPDF(<reportfile>, <outputfile>, [papersize], [dpu])
Parameters:	
<reportfile>	The file name of the report to convert.
<outputfile>	The output file to write.
[papersize]	The papersize to use.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPCL(ReportFile.rptx, OutFile.PDF, paperLetter)
Notes:	Converts the external Report file named <reportfile> to PDF. The result will be written to the file specified in <outputfile>. The [papersize] parameter is optional and specified the paper size to use. The default is Letter size (paperLetter). The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertFileToPNG	
Syntax:	ReportConvertFileToPNG(<reportfile>, <outputfile>, [dpu])
Parameters:	
<reportfile>	The file name of the report to convert.
<outputfile>	The output file to write.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPNG(ReportFile.rptx, OutFile.PNG)
Notes:	Converts the external Report file named <reportfile> to PNG. The result will be written to the file specified in <outputfile>. The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertFileToZPL	
Syntax:	ReportConvertFileToZPL(<reportfile>, <outputfile>, <heat>, <speed>, <printmode>, <mediatracking>, <mediatype>, [dpu], [extracommands])
Parameters:	
<reportfile>	The file name of the report to convert.
<outputfile>	The output file to write.
<heat>	The print heat to use, from -30 to 30.
<speed>	The print speed to use, from 3 to 6.
<printmode>	The Print mode: T for Tear off, P for Peel off, or R for Rewind
<mediatracking>	The Media Tracking: Y for Die Cut stock, or N for Continuous stock
<mediatype>	The Media Type: T for Thermal Transfer or D for Direct Thermal.
[dpu]	Optional. The Dots Per Unit to use when converting.
[extracommands]	Any raw ZPL commands that you need. These will go after the header commands and before the fields.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPNG(ReportFile.rptx, OutFile.PNG)
Notes:	Converts the external Report file named <reportfile> to ZPL. The result will be written to the file specified in <outputfile>.

ReportConvertToHTML	
Syntax:	ReportConvertFileToHTML(<report>, <outputfile>, [dpu])
Parameters:	
<report>	The name of the report to convert.
<outputfile>	The output file to write.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToHTML(ReportFile.rptx, OutFile.html)
Notes:	Converts the external Report file named <reportfile> to HTML. The result will be written to the file specified in <outputfile>. The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertToPCL	
Syntax:	ReportConvertFileToPCL(<reportfile>, <outputfile>, [dpu])
Parameters:	
<reportfile>	The name of the report to convert.
<outputfile>	The output file to write.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPCL(ReportFile.rptx, OutFile.PCL)
Notes:	Converts the external Report file named <reportfile> to PCL. The result will be written to the file specified in <outputfile>. The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertToPDF	
Syntax:	ReportConvertFileToPDF(<report>, <outputfile>, [papersize], [dpu])
Parameters:	
<report>	The name of the report to convert.
<outputfile>	The output file to write.
[papersize]	The paper size to use.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertToPDF(ReportFile.rptx, OutFile.PDF, paperLetter)
Notes:	Converts the internal Report named <report> to PDF. The result will be written to the file specified in <outputfile>. The [papersize] parameter is optional and specified the paper size to use. The default is Letter size (paperLetter). The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertToPNG	
Syntax:	ReportConvertFileToPNG(<reportfile>, <outputfile>, [dpu])
Parameters:	
<reportfile>	The name of the report to convert.
<outputfile>	The output file to write.
[dpu]	Optional. The Dots Per Unit to use when converting.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPNG(ReportFile.rptx, OutFile.PNG)
Notes:	Converts the external Report file named <reportfile> to PNG. The result will be written to the file specified in <outputfile>. The [dpu] parameter is optional and specifies the Dots Per Unit to use when calculating sizes. This parameter should be left out or set to 0 in most cases.

ReportConvertToZPL	
Syntax:	ReportConvertFileToZPL(<reportfile>, <outputfile>, <heat>, <speed>, <printmode>, <mediatracking>, <mediatype>, [dpu], [extracommands])
Parameters:	
<reportfile>	The name of the report to convert.
<outputfile>	The output file to write.
<heat>	The print heat to use, from -30 to 30.
<speed>	The print speed to use, from 3 to 6.
<printmode>	The Print mode: T for Tear off, P for Peel off, or R for Rewind
<mediatracking>	The Media Tracking: Y for Die Cut stock, or N for Continuous stock
<mediatype>	The Media Type: T for Thermal Transfer or D for Direct Thermal.
[dpu]	Optional. The Dots Per Unit to use when converting.
[extracommands]	Any raw ZPL commands that you need. These will go after the header commands and before the fields.
Returns:	Returns 1 if the conversion was successful, or 0 if not.
Examples:	ReportConvertFileToPNG(ReportFile.rptx, OutFile.PNG)
Notes:	Converts the external Report file named <reportfile> to ZPL. The result will be written to the file specified in <outputfile>.

6.14 Response Functions

AcceptPrompt	
Syntax:	AcceptPrompt( )
Parameters:	None
Returns:	Nothing
Examples:	AcceptPrompt( )
Notes:	Causes a prompt to be Accepted, exactly as if an Accept Button was clicked. All of the Validation scripts and the Next Prompt scripts are evaluated. ValidationFail will stop the Accept process.

CancelPrompt	
Syntax:	CancelPrompt( )
Parameters:	None
Returns:	Nothing
Examples:	CancelPrompt( )
Notes:	Causes a prompt to be Exited, exactly as if an Exit Button was clicked.

ResponseScan	
Syntax:	ResponseScan()
Parameters:	None
Returns:	Returns the raw data from the last successful barcode scan.
Examples:	@scan@ = ResponseScan( )
Notes:	This function would usually be used in the AfterScan event, but is valid in any In-Prompt script.

ResponseSource	
Syntax:	ResponseSource()
Parameters:	None
Returns:	Returns the Source of the last input. Valid sources are srcKeyboard, srcScanner, srcImage, or srcText.
Examples:	ResponseSource() returns srcScanner if the last input was scanned.
Notes:	Returns the Source of the last input.

ResponseSymbology	
Syntax:	ResponseSymbology()
Parameters:	None
Returns:	Returns the Symbology of the barcode scanned for the last response.
Examples:	ResponseSymbology() returns 1 (bcCode39) if the last barcode scanned was Code 39.
Notes:	Returns the barcode symbology of the last scan. See the Constants list for the symbology IDs.

ValidationFail	
Syntax:	ValidationFail(<String>, [element])
Parameters:	
<String>	The error message to be displayed on the device display
[element]	Optional. The name of the element to receive the keyboard focus.
Returns:	Nothing
Examples:	ValidationFail("Error", “prompt1.textbox1”)
Notes:	This function may be used in the After Prompting and After Validation scripts. Call to cause the validation of this response to fail. The device will display the message you place in <String>. If this function is called multiple times in a script, the message from the first occurrence will be displayed. If a multi-prompt element name is specified in [element], the keyboard focus will be set to this element when the notification message is closed.

6.15 Serial Functions

BTClose	
Syntax:	BTClose(<bluetoothname>)
Parameters:	
<bluetoothname>	The name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection.
Returns:	Nothing
Examples:	BTClose("Bluetooth Device")
Notes:	Closes the Bluetooth SPP connection previously opened with BTOpen.

BTIsConnected	
Syntax:	BTIsConnected(<bluetoothname>)
Parameters:	
<bluetoothname>	The name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection.
Returns:	Returns 1 if the Bluetooth SPP device is connected, or 0 if not.
Examples:	@erc@ = BTIsConnected("Bluetooth Device")
Notes:	Returns the connection status of the Bluetooth SPP connection specified.

BTOpen	
Syntax:	BTOpen(<bluetoothname>)
Parameters:	
<bluetoothname>	The name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection.
Returns:	Returns 1 if the port was opened, or 0 if not.
Examples:	@erc@ = BTOpen("Bluetooth Device")
Notes:	Opens the Bluetooth SPP connection specified.

BTRead	
Syntax:	BTRead(<bluetoothname> , <Timeout> , <Endofline>)
Parameters:	
<bluetoothname>	The name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection.
<Timeout>	The device will wait <Timeout> milliseconds before timing out.
<Endofline>	Data from the port will be read until the <Endofline> is read
Returns:	The data that is read from the port
Examples:	@Data@ = BTRead("Bluetooth Device", 5000, ascCR)
Notes:	Writes the data in <String> to <Port> using the <Baud>, <Parity> and <Bits> specified. Non-printable ASCII characters will be encoded as [XXX] where XXX is the 3-digit decimal code for the character. The device will wait <Timeout> milliseconds before timing out. A timeout of 0 will cause the timeout to be infinite. A Timeout between 1 and 100ms will not display the status message while waiting. This allows silent polling of the serial port by a timer function. The Escape key will terminate the function.Reads from the Bluetooth SPP connection specified in <port>. The resulting data is returned. The device reads until the character specified in <endofline> is read. The <bluetoothname> is the name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection. The device will wait <timeout> milliseconds before timing out. Non-printable ASCII characters will be encoded as [XXX] where XXX is the 3-digit decimal code for the character.

BTWrite	
Syntax:	BTWrite(<bluetoothname> , <String>)
Parameters:	
<bluetoothname>	The name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection.
<String>	Data to be written to the serial port
Returns:	The number of characters written to the port
Examples:	@Written@ = BTWrite("Bluetooth Device", "12345" & ascCR & ascLF)
Notes:	Writes the data in <string> to the Bluetooth SPP connection specified. The <bluetoothname> is the name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection. This function returns the number of characters written to the Bluetooth SPP connection. Non-printable ASCII characters can be encoded as [XXX] where XXX is the 3-digit decimal code for the character.

BTWriteRead	
Syntax:	BTWriteRead(<bluetoothname> , <Timeout> , <String>, <EndOfLine>)
Parameters:	
<bluetoothname>	The name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection.
<Timeout>	The device will wait <Timeout> milliseconds before timing out.
<String>	Data to be written to the serial port
<EndOfLine>	Data from the port will be read until the <Endofline> is read
Returns:	The data read from the port
Examples:	@Data@ = BTWriteRead("Bluetooth Device", 1000, "12345" & ascCR & ascLF, ascCR)
Notes:	Writes the data in <string> to the Bluetooth SPP connection specified. Then, reads the port until the character specified in <endofline> is read. The <bluetoothname> is the name or MAC address of a paired Bluetooth device that supports the Bluetooth SPP connection. The device will wait <timeout> milliseconds before timing out. This function returns the data read from the Bluetooth SPP connection. Non-printable ASCII characters can be encoded as [XXX] where XXX is the 3-digit decimal code for the character. Returned Non-printable ASCII data will also be encoded.

SerialClose	
Syntax:	SerialClose(<Port>)
Parameters:	
<Port>	Number indicating the serial port to close. This can be in the range serCOM1 to serCOM9.
Returns:	Nothing
Examples:	SerialClose(serCOM1)
Notes:	Closes the serial port previously opened with SerialOpen.

SerialFlush	
Syntax:	SerialFlush(<Port>)
Parameters:	
<Port>	Number indicating the serial port to flush. This can be in the range serCOM1 to serCOM9.
Returns:	Nothing
Examples:	SerialFlush (serCOM1)
Notes:	Flushes any received data that has not been read from the port.

SerialOpen	
Syntax:	SerialOpen(<Port>, <baud>, <parity>, <bits>)
Parameters:	
<Port>	Number indicating the serial port to open. This can be in the range serCOM1 to serCOM9.
<baud>	The baud rate to use.
<parity>	Indicate Odd, Even, or No parity.
<bits>	The number of data bits to use for communications.
Returns:	Returns 1 if the port was opened, or 0 if not.
Examples:	@ret@ = SerialOpen(serCOM1, serBaud9600, serParityNone, serData8)
Notes:	Once the port has been opened, it will remain open until closed by SerialClose or until the data collection program is exited.

SerialRead	
Syntax:	SerialRead(<Port> , <Baud> , <Parity> , <Bits> , <Timeout> , <Endofline>)
Parameters:	
<Port>	Number indicating the COM port to read
<Baud>	Baud rate
<Parity>	Parity setting
<Bits>	Number of data bits
<Timeout>	The device will wait <Timeout> milliseconds before timing out.
<Endofline>	Data from the port will be read until the <Endofline> is read
Returns:	The data that is read from the port
Examples:	@Data@ = SerialRead(serCOM1, serBaud9600, serParityNone, serData8, 5000, ascCR)
Notes:	Writes the data in <String> to <Port> using the <Baud>, <Parity> and <Bits> specified. Non-printable ASCII characters will be encoded as [XXX] where XXX is the 3-digit decimal code for the character. The device will wait <Timeout> milliseconds before timing out. A timeout of 0 will cause the timeout to be infinite. A Timeout between 1 and 100ms will not display the status message while waiting. This allows silent polling of the serial port by a timer function. The Escape key will terminate the function.

SerialWrite	
Syntax:	SerialWrite(<Port> , <Baud> , <Parity> , <Bits> , <Timeout> , <String>)
Parameters:	
<Port>	Number indicating the COM port
<Baud>	Baud rate
<Parity>	Parity setting
<Bits>	Number of data bits
<Timeout>	The device will wait <Timeout> milliseconds before timing out.
<String>	Data to be written to the serial port
Returns:	The number of characters written to the port
Examples:	@Written@ = SerialWrite(serCOM1, SerBaud9600, serParityNone, serData8, 1000, "12345" & ascCR & ascLF)
Notes:	Writes the data in <String> to <Port> using the <Baud>, <Parity> and <Bits> specified. Non-printable ASCII characters can be encoded as [XXX] where XXX is the 3-digit decimal code for the character.

SerialWriteRead	
Syntax:	SerialWriteRead(<Port> , <Baud> , <Parity> , <Bits> , <Timeout> , <String>, <EndOfLine>)
Parameters:	
<Port>	Number indicating the COM port
<Baud>	Baud rate
<Parity>	Parity setting
<Bits>	Number of data bits
<Timeout>	The device will wait <Timeout> milliseconds before timing out.
<String>	Data to be written to the serial port
<EndOfLine>	Data from the port will be read until the <Endofline> is read
Returns:	The data read from the port
Examples:	@Data@ = SerialWriteRead(serCOM1, serBaud9600, serParityNone, serData8, 1000, "12345" & ascCR & ascLF & ascCR)
Notes:	Writes the data in <String> to <Port> using the <Baud>, <Parity> and <Bits> specified. Then, reads the port until the character specified in <Endofline> is read. The device will wait <Timeout> milliseconds before timing out. A timeout of 0 will cause the timeout to be infinite. A Timeout between 1 and 100ms will not display the status message while waiting. This allows silent polling of the serial port by a timer function. The Escape key will terminate the function. This function returns the data read from the port. Non-printable ASCII characters can be encoded as [XXX] where XXX is the 3-digit decimal code for the character. Returned non-printable ASCII data will also be encoded.

6.16 String Functions

EndsWith	
Syntax:	EndsWith(<String> , <EndsWith>, [CaseSensitive])
Parameters:	
<String>	The first string to compare against.
<EndsWith>	The string to test.
[CaseSensitive]	Optional. If 1 the comparison will be case sensitive.
Returns:	Returns 1 if <String> ends with <EndsWith>, returns 0 otherwise.
Examples:	@ret@ = EndsWith("Hello World", "World")
Notes:	If [CaseSensitive] is 0, the comparison will be case-insensitive. If [CaseSensitive] is non-zero, the comparison will be case-sensitive.

Exact	
Syntax:	Exact(<String1> , <String2>)
Parameters:	
<String1>	The first string to test
<String2>	The second string to test
Returns:	Returns 1 if <string1> exactly matches <string2> (except for case), returns 0 otherwise.
Examples:	Exact(@MyString@,"Hello") = 1 if @MyString@ is a variable with the value of "Hello"
Notes:	The string comparison is case insensitive.

InStr	
Syntax:	InStr(<Start>, <String> , <SearchFor>, <CaseSensitive>)
Parameters:	
<Start>	Starting position to search <String>. The first character is position 1.
<String>	The string to search
<SearchFor>	The string to search for within <String>
<CaseSensitive>	1 to perform a case-sensitive search; 0 ignores case
Returns:	Returns the position of the first match from the start position of the first character in the SearchFor string. Returns 0 if SearchFor string is not found.
Examples:	InStr(1,"Hello World",”o",1) returns 5.
Notes:	Locates the string <SearchFor> within <String>. If found, the position of the first character of the matching string is returned.

InStrRev	
Syntax:	InStrRev(<Start>, <String> , <SearchFor>, <CaseSensitive>)
Parameters:	
<Start>	The character position to start searching from
<String>	The string to search
<SearchFor>	The string to search for within <String>
<CaseSensitive>	Whether to perform a case-sensitive search
Returns:	The character position of the first character of the matching string in <String>.
Examples:	InStrRev(1,"Hello World","o",1) returns 8
Notes:	Same as InStr, but the search works from right to left. The search always starts from the right end of the string, but stops searching at <Start>.

IsNumeric	
Syntax:	IsNumeric(<Expression>)
Parameters:	
<Expression>	A string containing the characters to be tested.
Returns:	Returns 1 (same as TRUE) if the expression is numeric (contains no alpha characters), else 0 (same as FALSE).
Examples:	IsNumeric("7") returns TRUE
Notes:	The digits 0 - 9 and '.' are considered numeric. Leading plus or minus signs are allowed. Everything else causes a FALSE return.

LCase	
Syntax:	LCase(<String>)
Parameters:	
<String>	The string to be converted to lowercase
Returns:	Returns <String> converted to all lowercase
Examples:	LCase("Hello") returns "hello".
Notes:	All alphabetic characters are converted to lowercase. Numeric and punctuation characters are not changed.

Left	
Syntax:	Left(<String> , <Num>)
Parameters:	
<String>	The string from which the characters should be returned
<Num>	The number of characters to be returned
Returns:	Returns a substring of <String> containing the left <Num> number of characters.
Examples:	Left("Hello",2) returns "He"
Notes:	The <String> parameter may be a string or numeric expression.

Len	
Syntax:	Len(<String>)
Parameters:	
<String>	The string whose length is to be calculated.
Returns:	Returns the number of characters in <String>.
Examples:	Len("Hello") returns 5.
Notes:	The <String> parameter may be a string or numeric expression.

LTrim	
Syntax:	LTrim(<String>)
Parameters:	
<String>	The string to trim
Returns:	Returns <String> with all leading spaces removed.
Examples:	LTrim(" Hello ") returns "Hello ".
Notes:	The <String> parameter may be a string or numeric expression.

Mid	
Syntax:	Mid(<String> , <Start>, <Num>)
Parameters:	
<String>	The string from which the characters should be returned.
<Start>	The starting position within the string. The first character is position 1.
<Num>	The number of characters to return
Returns:	Returns a substring of <String> starting with the <Start> position with a length of <Num> characters.
Examples:	Mid("Hello", 2, 3) returns "ell".
Notes:	The <String> parameter may be a string or numeric expression.

Pad	
Syntax:	Pad(<String>, <Length>)
Parameters:	
<String>	The string to be padded
<Length>	The number of characters of the returned string
Returns:	The <String> padded with spaces to the length <Length>
Examples:	Pad("Hello",10) returns "Hello ".
Notes:	Adds trailing spaces to force the string to be the length specified.

PadLeft	
Syntax:	PadLeft(<String>, <Length>, <PadChar>)
Parameters:	
<String>	The string to be padded
<Length>	The number of characters of the returned string
<PadChar>	The character to pad with.
Returns:	The <String> padded to the length <Length> with <PadChar>
Examples:	PadLeft("123.45",10, "0") returns "0000123.45".
Notes:	Adds padding characters to the left of the string.

PadRight	
Syntax:	PadRight(<String>, <Length>, <PadChar>)
Parameters:	
<String>	The string to be padded
<Length>	The number of characters of the returned string
<PadChar>	The character to pad with.
Returns:	The <String> padded to the length <Length> with <PadChar>
Examples:	PadLeft("123.45",10, "0") returns "123.450000".
Notes:	Adds padding characters to the right of the string.

Replace	
Syntax:	Replace(<String>, <Replace>, <ReplaceWith>, <CaseSensitive>)
Parameters:	
<String>	The string in which replacements are to be made
<Replace>	The string to replace
<ReplaceWith>	The string to replace with
<CaseSensitive>	Whether to perform a case sensitive search
Returns:	Returns the string with <Replace> replaced with <ReplaceWith>
Examples:	Replace("ABC 123 xyz", "123", "9", 0) returns "ABC 9 xyz"
Notes:	Set <CaseSensitive> to 1 to enable case-sensitive replacement, else set it to 0 to allow replacement regardless of case. This function replaces all instances of the <Replace> string.

Rept	
Syntax:	Rept(<String>, <Number>)
Parameters:	
<String>	The string to repeat
<Number>	The number of times to repeat the string
Returns:	Returns a string that contains <String> repeated <Number> times
Examples:	Rept("Hello",3) returns "HelloHelloHello"
Notes:	The parameter <String> may be string or numeric data. The <Number> parameter will be treated as an integer.

Right	
Syntax:	Right(<String> , <Num>)
Parameters:	
<String>	The string from which the characters should be returned
<Num>	The number of characters to be returned
Returns:	Returns a substring of <String> containing the right <Num> of characters
Examples:	Right("Hello",2) returns "lo"
Notes:	The <String> parameter may be a string or numeric expression.

RTrim	
Syntax:	RTrim(<String>)
Parameters:	
<String>	The string to be trimmed
Returns:	Returns <String> with any trailing spaces removed
Examples:	RTrim(" Hello ") returns " Hello"
Notes:	Trims trailing spaces

Search	
Syntax:	Search(<String> , <Search> , <StartPos>)
Parameters:	
<String>	The string to search
<Search>	The characters to search for
<StartPos>	The starting position within <String> to search.
Returns:	Returns the position that any character from <Search> appears within <String>, starting from <StartPos>. If no character from <Search> is found in <String>, the length of the string is returned.
Examples:	Search("Hello World", "eo", 1) returns 2.
Notes:	Searches for any one of a set of characters within a string.

Space	
Syntax:	Space(<Length>)
Parameters:	
<Length>	The number of spaces to format into the string
Returns:	Returns a string containing spaces of the length specified
Examples:	Space(5) returns " "
Notes:	Creates a string of <Length> spaces

Split	
Syntax:	Split(<String> , <Char>, <Result1>, <Result2>, ...)
Parameters:	
<String>	The string to split
<Char>	The character at which to split.
<Result1>	The string to receive the first substring
<Result2>	The string to receive the second substring
	You may specify any number of result strings to receive the data.
Returns:	Returns the number of strings parsed. The data will be in Result1, Result2, etc.
Examples:	Split("Hello World"," ",Result1, Result2) returns 2 and puts "Hello" in Result 1 and "World" in Result2.
Notes:	Parse the <String> at the character specified by <Char> into the result strings <Result1>, <Result2>, etc. Any number of result strings may be specified.

SplitN	
Syntax:	SplitN(<String> , <Char>, <Index>)
Parameters:	
<String>	The string to split
<Char>	The character at which to split.
<Index>	Specifies which substring to return.
Returns:	Returns the value in the <Index> position.
Examples:	SplitN("ABC!DEF!GHI","!", 2) returns "DEF".
Notes:	Parse <String> using the <Char> as the separator, returning the value in the <Index> position. The first value is referenced by <Index> 1.

StartsWith	
Syntax:	StartsWith(<String> , <StartsWith>, [CaseSensitive])
Parameters:	
<String>	The string to search.
<StartsWith>	The string to compare with.
[CaseSensitive]	Specifies whether the search should be case-sensitive.
Returns:	Returns 1 if <String> starts with <StartsWith>, returns 0 otherwise.
Examples:	@ret@ = StartsWith("Hello World", "He")
Notes:	If [CaseSensitive] is 0, the comparison will be case-insensitive. If [CaseSensitive] is non-zero, the comparison will be case-sensitive.

StrComp	
Syntax:	StrComp(<String1> , <String2>)
Parameters:	
<String1>	The first string to compare
<String2>	The second string to compare
Returns:	Returns 1 if <String1> is greater (sorts after) <String2>. Returns 0 if <String1> and <String2> match. Returns -1 if <String1> is less than (sorts before) <String2>.
Examples:	StrComp("Hello","World") returns -1
Notes:	This comparison is case-sensitive.

StrDup	
Syntax:	StrDup(<Length> , <String>)
Parameters:	
<Length>	The number of times to repeat <String>
<String>	The string to be repeated
Returns:	Returns a string that contains the first character of <String> repeated <Length> times.
Examples:	StrDup(5, "Hello") returns "HHHHH"
Notes:	The <String> parameter may be a string or numeric expression.

Subst	
Syntax:	Subst(<String>, <Start>, <Length>, <Replace>)
Parameters:	
<String>	The string to replace in
<Start>	The starting position of the replacement within <string>
<Length>	The length to replace
<Replace>	The replacement string
Returns:	Returns <String> with the <Length> number of characters from the <Start> position replaced by the string in <Replace>.
Examples:	Subst("Collect Asset Data", 9, 5, "Inventory") returns "Collect Inven Data".
Notes:	Both <String> and <Replace> may be string or numeric expressions.

Text	
Syntax:	Text(<Data>, <Mask>)
Parameters:	
<Data>	The source data to be formatted
<Mask>	The mask to use for formatting the text.
Returns:	Returns the data formatted according to the mask
Examples:	Text("4405551212","(???) ???-????") = "(440) 555-1212"
Notes:	The Mask character is ?. Any other character is literal. You may use the escape char: \, i.e. \? = literal ?.

Trim	
Syntax:	Trim(<String>)
Parameters:	
<String>	The string to be trimmed
Returns:	The <String> with all leading and trailing spaces removed
Examples:	Trim(" Hello ") returns "Hello"
Notes:	Trims leading and trailing spaces

UCase	
Syntax:	UCase(<String>)
Parameters:	
<String>	The string to be converted to uppercase
Returns:	Returns <String> will all alpha characters converted to uppercase
Examples:	UCase("Hello") returns "HELLO”.
Notes:	Converts the string to all uppercase

6.17 Keywords

IF	
Syntax:	IF(<expression>)
Parameters:	
<expression>	Logical expression to be evaluated

Examples:	IF(@UserVar@ = 1)
	   @Count@ = @Count@ + 1
	ENDIF

Notes:	An IF Statement. This can take the form:
	IF(<expression>)
	   Statements
	ELSEIF(<expression>)
	   Statements
	ELSE
	   Statements
	ENDIF

ELSE	
Syntax:	ELSE
Parameters:	None

Examples:	IF(@UserVar@ = 1)
	   @Count@ = @Count@ + 1
	ELSE
	   @UserVar@ = 0
	   Message(“Tap OK to Continue”,“User Message”,”OK”,””
	ENDIF

Notes:	See IF

ELSEIF	
Syntax:	ELSEIF(<expression>)
Parameters:	
<expression>	Logical expression to be evaluated

Examples:	IF(@UserVar@ = 1)
	   @Count@ = @Count@ + 1
	ELSEIF (@UserVar@ = 0)
	   @Count@ = @Count@ - 1
	ENDIF

Notes:	See IF

ENDIF	
Syntax:	ENDIF
Parameters:	None
Examples:	See IF
Notes:	See IF

FOR	
Syntax:	FOR(<init>, <expression>, <update>)
Parameters:	
< init >	Statement to initialize the FOR loop
< expression >	If the expression evaluates to TRUE, the statements in the FOR loop execute. If the expression evaluates to FALSE, the FOR loop is done
< update >	The statement that increments the looping variable, the update executes after the statements in the FOR loop execute for its current pass through the loop.

Examples:	FOR(@Var@=1,@Var@<10,@Var@=@Var@+1)
	   Message("Next Count: " & @Count@,"User Message","OK","")
	NEXT

Notes:	A FOR Loop. This can take the form:
	FOR(<init>, <expression>, <update>)
	   Statements
	NEXT

NEXT	
Syntax:	NEXT
Parameters:	None
Examples:	See FOR
Notes:	See FOR

EXITFOR	
Syntax:	EXITFOR
Parameters:	None

Examples:	FOR(@Var@=1,@Var@<10,@Var@=@Var@+1)
	   @Msg@ = Message("Count: " & @Var@,"User Message","OK","Stop")
	   IF (@Msg@ = 2)
	      EXITFOR
	   ENDIF
	NEXT

Notes:	Exits a FOR Loop

WHILE	
Syntax:	WHILE(<expression>)
Parameters:	
<expression>	Logical expression to be evaluated

Examples:	@Count@ = 0
	WHILE(@Count@ < 10)
	   @Count@ = @Count@ + 1
	   Message(“Count: “ & @Count@,”User Message”,”OK”,””)
	WEND

Notes:	A WHILE Loop. This can take the form:
	WHILE(<expression>)
	   Statements
	WEND

WEND	
Syntax:	WEND
Parameters:	None
Examples:	See WHILE
Notes:	See WHILE

EXITWHILE	
Syntax:	EXITWHILE
Parameters:	None

Examples:	WHILE(TRUE)
	   IF (@Count@ => 10)
	      EXITWHILE
	   ENDIF
	   @Count@ = @Count@ + 1
	WEND

Notes:	Exits a WHILE loop

EXIT	
Syntax:	EXIT
Parameters:	None

Examples:	FOR(@Var@=1,@Var@<10,@Var@=@Var@+1)
	   @Msg@ = Message("Count: " & @Var@,"User Message","OK","Stop")
	   IF (@Msg@ = 2)
	      EXIT
	   ENDIF
	NEXT

Notes:	Exits a Script

Index