CETerm Scripting Guide

for Version 5.5.0 or later

Naurtech Web Browser
And
Terminal Emulation Clients

For Windows CE Devices

CETerm | CE3270 | CE5250 | CEVT220

Copyright Notice

Trademarks

Software Version

This document is for Version 5.5.0 or later of Naurtech Web Browser and Terminal Emulation clients.

Table of Contents

Preface

All of us at Naurtech Corporation constantly strive to deliver the highest quality products and services to our customers. We are always looking for ways to improve our solutions. If you have comments or suggestions, please direct these to:

Naurtech Corporation
e-mail: support@naurtech.com
Phone: +1 (425) 837.0800

Assumptions

This manual assumes you have a working knowledge of:

- Microsoft Windows user interface metaphor and terminology.
- Stylus based touch screen navigation terminology.
- Basic programming and scripting concepts.
- Dynamic HTML, the browser DOM, and JavaScript.
- Basic operations and requirements of the host applications you want to access with the Naurtech web browser and terminal emulation clients.

Conventions used in this Manual

This manual uses the following typographical conventions:

- All user actions and interactions with the application are in bold, as in **[Session][Configure]**
- Any precautionary notes or tips are presented as follows

  Tip: Text associated with a specific tip

- ☞ represents new version specific information
- All text associated with samples is presented as follows.

```
/*alert*/
OS.Alert("Script done.");
```

Additional Documentation

Naurtech Scripting is an integral feature of Naurtech Web Browser and Terminal Emulation Clients. Please refer to the User's Manual for detailed installation and configuration information. The User's Manual may be downloaded from the "Support" section of www.naurtech.com.

Online Knowledgebase

Although we continually strive to keep this manual up to date, you may find our online support knowledgebase useful for the latest issues, troubleshooting tips and bug fixes. You can access the support knowledgebase from our website at:

www.naurtech.com → Support → Knowledgebase

1.0 Introduction

The Naurtech CETerm Clients provide a robust and flexible environment for Terminal Emulation and Web based applications on a mobile device. Our Clients are available for most Windows CE platforms including CE .NET 4.2, Windows CE 5.0, Windows CE 6.0, Windows Mobile 2003, Windows Mobile 5, and Windows Mobile 6.

Device tailored versions of our Clients are available for most industrial terminals. These versions integrate the peripherals on each device, such as the barcode scanner, magnetic stripe reader, RFID reader and Bluetooth printer. Naurtech Scripting features provide additional control of these peripherals and simplify tasks such as data collection, validation, and automation.

All Naurtech Clients include one or more Terminal Emulations (TE) and a Web Browser for a natural migration path from legacy text based TE applications to newer Web based applications. We will refer to the clients collectively as CETerm, although the scripting features apply fully to the single emulation products CE3270, CE5250, and CEVT220.

Scripting features can help the transition to web applications and add capabilities to older TE applications. Newer web based applications can be presented in a familiar single-purpose (locked down) configuration which uses keys, the touch screen, or both for user interactions. Please see our “Web Browser Programming Guide” for detailed information on using the Web Browser features.

The Naurtech Scripting features automate and extend our clients. We use the industry standard JavaScript language with Microsoft JScript additions. JavaScript is the language underlying the most capable and complex functionality available in web applications today. This new class of web applications is sometimes referred to as “Web 2.0” using Asynchronous JavaScript and XML (AJAX). CETerm brings this mature and rich language to the TE user to provide more productive TE applications. Scripting can also interact with web browser sessions to enrich and extend existing web applications on the mobile device.

Scripts can be as simple as editing barcode data before sending to a host or as complex as parsing an external XML document, applying an XSLT transformation and returning the result to the host through the TE session. CETerm Automation Objects are provided to give scripts access to the state of CETerm, the TE session, hardware components, and access to Windows CE operating system functions such as network, file and registry operations.

This guide is intended to describe the steps for writing and running scripts and the features provided through the CETerm Automation Objects. Please consult the standard references for details on JavaScript (or JScript) syntax and XML.

You may also need to consult standard references for HTML syntax, the browser Document Object Model (DOM), and other aspects of Dynamic HTML if you are scripting web browser features. Please refer to the Naurtech User's Manual for details on basic usage and configuration of the Naurtech clients.

We hope that our Scripting features will enrich and extend the capabilities of your TE and browser applications. Explore a little deeper and we think you will be amazed at the possibilities for building powerful business applications.

1.1 FEATURE HIGHLIGHTS

Following are some of the special features in Naurtech Scripting

- **JavaScript.** Naurtech uses the industry standard JavaScript scripting language. This powerful language is familiar to programmers and non-programmers world-wide as the core of rich web applications. With JScript, the Microsoft version of JavaScript, additional features are available such as the ability to use ActiveX objects in scripts.

- **On-device Script Editing.** Scripts are saved within the CETerm configuration and can be edited and tested right on the mobile device. Scripts can be imported and exported via text files on the device as well as loaded dynamically from files.

- **Cross Session Scripting.** All Naurtech clients allow up to 5 simultaneous sessions. Scripts can access and control any or all sessions. For example, you could extract text from one TE session and insert it into a different TE session or into a Web application.

- **Automation Objects.** CETerm Automation Objects are available to access and control the state of CETerm, the state of a TE or web browser session, the mobile device, and the Windows CE Operating System. Together these objects provide a rich set of features to simplify routine steps or build complex applications. For example, you can use an automation object to examine the current screen contents to trigger special actions.

- **Enriched Web Browser Applications.** Naurtech Scripting can interact with a web browser session to enrich existing web applications that were not written for a mobile device. For example, key bindings can be added to activate items in the page and scanned barcode or RFID data can be directed to input elements.

- **Workflow Automation.** Scripts can be used to automate routine tasks. The task may be a simple login process or a complex set of steps in your host application.

- **Event Activated Scripts.** There are several events within CETerm that will run associated scripts. For example, when a barcode is read, the script "OnBarcodeRead" will execute and will allow arbitrary processing of the barcode data before it is submitted to the TE or web browser session.

- **Key, Button, and Menu Activated Scripts.** Like most other CETerm actions, scripts can also be tied to any key combination, a toolbar button, or a context menu.

- **Timer Activated Scripts.** Scripts can be scheduled to run at a future time or run periodically.

- **Host Activated Scripts.** Host applications can also invoke scripts using special commands within the TE data stream.

2.0 Getting Started

This section describes some common ways that scripting features can be used within CETerm. Here we describe the JavaScript engine in CETerm and show how to load and edit a script. We also show sample scripts which (1) handle scanner input, (2) auto-login a terminal emulation session, and (3) provide user-specified "hot-spots" on the screen. Only small code "snippets" are shown. For complete details see the later sections of this manual.

2.1 JAVASCRIPT ENGINE

The CETerm JavaScript engine is a full JavaScript environment running in CETerm that provides all the power and familiarity of JavaScript for automating and extending your data collection process. Strictly speaking, CETerm contains the Microsoft JScript engine, which has additional capabilities, but we will refer to it as JavaScript.

The CETerm JavaScript engine is separate from the JavaScript engines which are available in web browser sessions, but the two engines can communicate, exchange data and send commands. Unlike the web browser engine, the CETerm engine runs independently of any TE or browser session and can interact with all sessions. This persistence allows the CETerm engine to maintain state throughout a data collection workflow.

The CETerm script engine runs as part of the CETerm user interface and when processing a script, the device keys and screen may be unresponsive. Think of the script engine as a virtual user which can examine the screen and send input. There are several techniques to write asynchronous scripts and to show feedback to the user and get user input while a script is running.

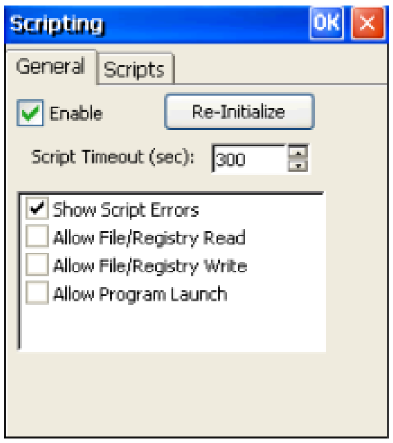
2.2 Enabling Scripting and Editing Scripts

Scripting is disabled by default. To enable scripting, open the configuration dialog

```
[Session]->[Configure]->[Options]->[Configure Scripting]
```

General Settings

On the **General** tab, check the **Enable** box and check **Show Script Errors**. You may also want to enable file and registry access permission or program launching if you need these features. The **Re-Initialize** button on this tab can be used if you have made changes to the permissions or your scripts and you wish to load the changes. The re-initialization does not take place until the dialog is closed.

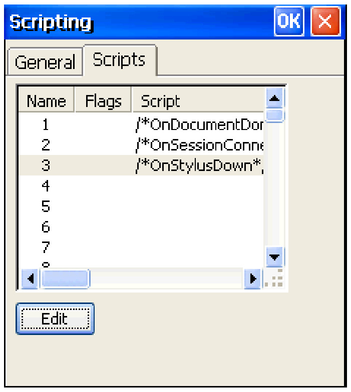


[✓] Enable
Script Timeout (sec): 300
[✓] Show Script Errors
[ ] Allow File/Registry Read
[ ] Allow File/Registry Write
[ ] Allow Program Launch

The **Script Timeout** variable limits the duration of script execution. This limit is useful when developing new scripts and as a safeguard against a script with an “infinite loop”. A value of 0 will disable the timeout. During execution, a script can modify the timeout value and reset the timer to allow additional execution time.

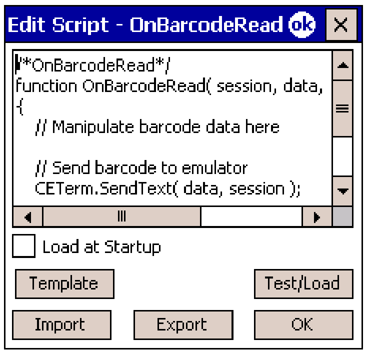
Editing Scripts

Scripts are edited on the **Scripts** tab. There are 64 script slots. The size of the script in each slot is limited to about 260,000 characters (about one-half megabyte under Windows CE). Scripts can also be loaded dynamically from files. A script slot will usually contain function definitions, which will be loaded into the engine, or executable statements such as function calls which may be bound to a key, toolbar, or menu.



Name	Flags	Script
1		/*OnDocumentDor
2		/*OnSessionConne
3		/*OnStylusDown*,
4		
5		
6		
7		

After selecting a script slot and tapping the **Edit** button, an Edit Script dialog will appear. The edit dialog allows **Import** and **Export** of scripts. For initial script development it may be easier to edit on your desktop PC, copy the script to the device, and **Import** the script. You can use any programming editor that does not insert text formatting commands. Even Notepad will work fine, but do not edit scripts with Microsoft Word. Smaller editing changes are easily made on the device.



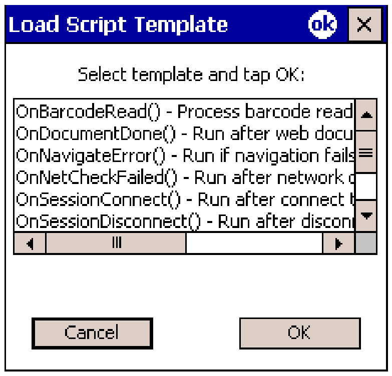
```
/*OnBarcodeRead*/
function OnBarcodeRead( session, data,
{
   // Manipulate barcode data here

   // Send barcode to emulator
   CETerm.SendText( data, session );
```

The checkbox **Load at Startup** should be checked for all scripts that contain function definitions that you want to have available in the script engine. The checkbox should **not** be checked for slots that contain scripts that are bound to keys or other activations. **Load at Startup** should be checked for all event handler definitions. All scripts marked **Load at Startup** will be loaded into the script engine when it starts with CETerm startup, or when **Re-Initialize** has been pressed on the **General** tab.

After importing or editing a script, you may want to tap the **Test/Load** button. If the script engine was previously enabled, the script will be executed. If the current script is a function definition, it will be checked for correct syntax and will be made available to the script engine. If the current script contains executable statements or is a function call, it will simulate activating the script. In general, you do not want to use **Test/Load** for executable statements.

Remember to tap **Test/Load** or **Re-Initialize** (with **Load at Startup** checked) after making changes to a script, if you want those changes loaded into the script engine. Also, **Test/Load** will not work if you have just checked **Enable** but not yet accepted the configuration changes.

The **Template** button displays a list of script templates which correspond to the scripting event handlers. Select a template and tap **OK** to have it replace the current contents of the script being edited. The template scripts show some of the ways to use CETerm Automation Objects.



Select template and tap OK:

OnBarcodeRead() - Process barcode read
OnDocumentDone() - Run after web docu
OnNavigateError() - Run if navigation fails
OnNetCheckFailed() - Run after network c
OnSessionConnect() - Run after connect t
OnSessionDisconnect() - Run after discon

2.3 CETerm Automation Objects

The CETerm Automation Objects provide access to the running CETerm application, session screens, device hardware, the Windows CE operating system, and other features. For example the command

```
   CETerm.PostIDA( "IDA_SESSION_S1", 0 );
```

within a script would switch CETerm to Session 1 if another session was currently active. Automation Objects can give access to the browser Document Object Model (DOM) of connected web sessions and the text on terminal emulation sessions. The IDA action codes are described briefly in the following section.

The CETerm Automation Objects are similar to ActiveX controls that are used in web pages, but they do not require any special creation operations prior to use. In fact, the same CETerm Automation Objects are accessible from both the CETerm JavaScript engine and the web browser JavaScript engines.

2.4 IDA Action Codes

An IDA Action Code is a special value that is used to invoke a device action, program action, or emulator action within the Naurtech Web Browser and Terminal Emulation Clients. IDA Action Codes can invoke special keys under terminal emulation, sound a tone, connect a session, or show the SIP. There are many IDA codes and these are documented in Appendix 1 of this manual. Almost any action which can be invoked by a KeyBar or assigned to a hardware key can be invoked by an IDA code. IDA codes can be submitted to CETerm in several different ways, under both scripting and the web browser.

2.5 OnBarcodeRead Script Event

CETerm generates several script events during operation. If there is a corresponding event handler defined within the CETerm script engine, then that handler will be invoked. The "OnBarcodeRead" event is a good example. The OnBarcodeRead event handler can intercept and pre-process barcode scan data using the full power of JavaScript before sending the data on to the TE or browser session.

The OnBarcodeRead handler could do something simple, such as pre-pending zero digits for short barcodes, or something complex such as splitting an Automotive Industry Action Group (AIAG) B-10 barcode and putting different parts into different fields on an IBM 5250 emulation screen.

Here is the OnBarcodeRead template that can be loaded in the script edit dialog

```
/* OnBarcodeRead */
function OnBarcodeRead ( session, data, source, type, date, time )
{
    // Manipulate barcode data here

    // Send barcode to emulator
    CETerm.SendText ( data, session );

    // Return 0 to handle barcode normally
    // Return 1 if handled data here
    return 1;
}
```

This handler simply passes the barcode data on to the current session using the “SendText” method. The return value of 1 tells CETerm not to pass on the barcode data with the usual wedge technique.

The following OnBarcodeRead handler will prefix 3 zeros to any 8 digit barcode and pass other barcodes unchanged

```
/* OnBarcodeRead */
function OnBarcodeRead ( session, data, source, type, date, time )
{
    // Prefix zeros to short barcodes
    if (data.length == 8)
    {
        data = "000" + data;
    }
    // Send barcode to emulator
    CETerm.SendText( data, session );

    // Return 0 to handle barcode normally
    // Return 1 if handled data here
    return 1;
}
```

If the OnBarcodeRead handler is defined, it will override any “ScannerNavigate” handler defined in a web page META tag. The following OnBarcodeRead handler will pass the scan on to the ScannerNavigate handler for a web browser in session 2

```
/* OnBarcodeRead */
function OnBarcodeRead ( session, data, source, type, date, time )
{
    // Don’t process for browser session
    if (session == 2)
    {
        // Return 0 to handle barcode with ScannerNavigate
        return 0;
    }
```

```
        // Prefix zeros to short barcodes
        if (data.length == 8)
        {
            data = "000" + data;
        }
        // Send barcode to emulator
        CETerm.SendText ( data, session );

        // Return 1 if handled data here
        return 1;
    }
```

The following OnBarcodeRead handler will split any barcode containing an ASCII Linefeed (LF = 0x0A) character and terminated with an ASCII ENQ (ENQ = 0x05) into two parts. The first part is put into the current IBM 5250 field and the second part into the next field and then submitted to the IBM host. This technique is used to login a user with a Code39 barcode in full-ASCII mode. All other barcodes are passed on for normal input

```
    /* OnBarcodeRead */
    function OnBarcodeRead ( session, data, source, type, date, time )
    {
        // Look for Full-ASCII Code 39 (LF = 0x0A) to mark Field Exit
        var lfIndex = data.indexOf( "\x0A" );

        if (lfIndex >= 0 )
        {
            var passwordStart = lfIndex + 1;

            // Look for Full-ASCII Code 39 (ENQ = 0x05)
            var enqIndex = data.lastIndexOf( "\x05" );

            if (enqIndex >= 0)
            {
                // NOTE: Using substr to extract user
                // Send User
                CETerm.SendText( data.substr( 0, lfIndex ), session );

                // Send field exit to advance cursor
                CETerm.SendIDA( "IDA_FIELD_EXIT", session );

                // Send Password
                // NOTE: Using substring to extract password
                CETerm.SendText( data.substring( passwordStart,
                                                 enqIndex ), session );

                // Submit form
                CETerm.SendIDA( "IDA_ENTER", session );

                // All scan data handled here
                return 1;
            }
        }

        // Handle scan data in normal way
        return 0;
    }
```

The `type` argument to OnBarcodeRead contains the labeltype of the barcode. This labeltype is related to the barcode symbology but usually a little more informative. The values are dependent on the hardware manufacturer but for most devices are the same as the Symbol LABELTYPE. The `type` is a small integer value representing a printable ASCII character (See Appendix 3). The `source` argument is the name of the scanner that read the barcode and is typically unused. The `date` and `time` are text strings representing the time of the read.

2.6 AUTOMATED LOGIN

Automating the host login process is a common task to speed workflow. CETerm contains a Macro record and playback that is usually used for this task. One limitation of the Macro feature is that it will only support a single session auto-connecting when CETerm starts. The scripting feature allows much more power and flexibility for automating the login or any complex or repetitive process.

Most auto-login features are based on a “prompt-and-response” mechanism that waits for text from the host (the prompt) and then sends some text (the response). The “expect” script and “ExpectMonitor” class provide the “prompt-and-response” mechanism within CETerm. The response is usually some simple text, but with the ExpectMonitor, it can be a script itself. The ExpectMonitor is also a good example of using script timers to perform long tasks. The full listing of the “expect” script and “ExpectMonitor” can be found in Section 5.1.

When “expect” is used for auto-login, it is activated within the “OnSessionConnect” event handler. Here is a simple example of an OnSessionConnect handler

```
    /* OnSessionConnect */
    function OnSessionConnect ( session )
    {
        // Set login information
        var myusername = "joeuser";
        var mypassword = "secret";
        var waittime = 8000;    // Milliseconds waiting for each text

        // Only login session 1
        if (session == 1)
        {
            // Look for "login" then "password"
            expect( session, waittime, "Login", myusername + "\r",
                                       "Password", mypassword + "\r" );
        }
    }
```

The expect arguments are `session` for the session index, `waittime` for the milliseconds waiting for each expected text, followed by pairs of expected text (prompt) and corresponding action(response). If the action is text, it is simply sent to the host when appropriate. There can be any number of (expected text, action) pairs as arguments. The expected text can be plain text or a regular expression.

For a case-insensitive match of “Login”, an appropriate regular expression could be `/login/i.` Regular expressions use the slash character as a delimiter rather than double-quote characters. The ‘i' indicates a case-insensitive match.

A more complex action can contain an anonymous function definition such as

```
    var beepMe = function (session) {CETerm.SendIDA ("IDA_BEEP_LOUD", 0);
                                     CETerm.SendText ("me\r", session );
    }
```

Combining these changes into the expect call would give

```
    expect( session, waittime, /login/i, beepMe,
                               "Password", mypassword + "\r" );
```

You might wonder why the `SendIDA` call in `beepMe` has a session index of 0 whereas `SendText` has the actual session argument. In this case we know that the beep action is not session specific and does not need to be sent to a specific session. In general, it is always OK to specify a session and it will be ignored for actions that do not require a value.

2.7 CUSTOM SCREEN HOT-SPOTS

A “hot-spot” is an area on a terminal emulation screen that is activated by taping with your finger or the stylus. CETerm supports several pre-defined hot-spots for TE sessions. With scripting, it is possible to define custom hot-spot behaviors. Custom hot-spots use the “OnStylusDown” event handler. Browser sessions do not support the OnStylusDown event because equivalent behavior can be implemented in HTML. You may need to disable the pre-defined hot-spots in CETerm because they will be triggered before a custom hot-spot.

The hot-spot action can depend on the screen contents in an area or simply be tied to a screen area. The following OnStylusDown handler can be loaded from the script templates

```
/* OnStylusDown */
function OnStylusDown( session, row, column )
{
    // Look for custom hot-spot
    var screen = CETerm.Session( session ).Screen;
    var text = screen.GetTextLine( row );
    if (text.match( /beep/i ))
    {
        OS.PlaySound( "default.wav", 0 );
    }
}
```

This hot-spot will play a sound if the line touched contains the word “beep”. The following hot-spots will activate VT function keys if the user touches in the specified rows and columns. In this case, the screen can show a box drawn with VT line drawing characters and text inside each box. With such a display, you can effectively create large glove-friendly on-screen buttons in TE.

```
/* OnStylusDown */
function OnStylusDown( session, row, column )
{
    // Buttons are on rows "start" through "end"
    var buttonrowstart = 9;
    var buttonrowend = 13;
    var IDA = "IDA_NONE";

    // Buttons are "buttonwidth" columns wide
    // Leftmost button is #1
    var buttonwidth = 5;
    var button = Math.floor((column + buttonwidth - 1) /
                              buttonwidth);

    if (row >= buttonrowstart && row <= buttonrowend)
    {
        switch (button) {
        case 1:   IDA = "IDA_VT_PF1"; break;
        case 2:   IDA = "IDA_VT_PF2"; break;
        case 3:   IDA = "IDA_VT_PF3"; break;
        case 4:   IDA = "IDA_VT_PF4"; break;
        }

        // DEBUG: Uncomment next two lines for testing
        //OS.Alert( "row=" + row + " col=" + column +
        //          " button=" + button + " IDA=" + IDA );

        if (!IDA.match("IDA_NONE"))
        {
            // Send command
            CETerm.PostIDA( IDA, session );
        }
    }
}
```

You may have noticed by now the use of `PostIDA` in some cases and `SendIDA` in other cases. `SendIDA` is a synchronous activation of an action whereas `PostIDA` is an asynchronous or deferred activation. In general it is always better to use `PostIDA` unless you **must** wait for the action to complete before proceeding in the script. The post action is similar to the "PostMessage" function in Windows programming and the send is similar to the "SendMessage" function. See the CETerm automation object for more details.

2.8 HANDLING JAVASCRIPT LITERAL VALUES

The Automation Objects described in Chapter 3 often return JavaScript literal values when the results are complex. For example, JavaScript literals are returned when the results are lists of files, lists of processes, or memory information.

A JavaScript literal is a text string which describes the contents of an array or object. These literals are easily converted into regular JavaScript arrays or objects for use by your scripts. This concept of data representation is similar to the highly popular "JavaScript Object Notation" (JSON, www.json.org) data interchange used by web applications. The results in CETerm do not follow the strict JSON format, but are handled in nearly identical ways.

WARNING: The format and content of JavaScript literals returned by various CETerm object may vary depending on hardware types or OS versions. They may also vary with different CETerm versions. You should review the raw form of the returned values on the devices you plan to use and program defensively to allow for variations. See the example below for details.

2.8.1 Array Literals

Here is a simple JavaScript statement that uses an array literal to create an array and assign it to a JavaScript variable:

```
   var myArray = [ 2, 4, 6 ];
```

myArray[2] now has the value 6.

If a CETerm object returned the same array literal, then you would use the JavaScript eval function to assign the array to a JavaScript variable:

```
   var arrayResult = "[2, 4, 6]";
   var myArray = eval( arrayResult );
```

myArray[0] now has the value 2.

2.8.2 Object Literals

Here is a simple JavaScript statement that uses an object literal to create an object and assign it to a JavaScript variable:

```
   var myObject = {name:'fred', attributes:0x21, size:12341234};
```

myObject.attributes has the value 0x21.

If a CETerm object returned the same object literal, then you would again use the JavaScript eval function to assign the object to a JavaScript variable. Because the curly-bracket operator may delimit either a block of statements or an object literal, you must convert object literal strings in a slightly different way:

```
   var objectResult = "{name:'fred', attributes:0x21, size:12341234}";
   var myObject = eval( "(" + objectResult + ")" ); // note parenthesis

   // or the preferred format
   eval( "var myObject=" + objectResult );
```

myObject.name now has the value 'fred'.

2.8.3 Complex Literals

A complex literal may consist of nested array and object literals, but it is treated in the same manner. In general you should use the eval syntax shown for the object literal for all types of array, object, or complex literals.

2.8.4 Optional Object Properties

The object literal returned as a result may not always contain all possible properties. This is true for the File.GetList() method. The various file timestamps are not always present. Your JavaScript code must check these values or be prepared to handle the “undefined” value that can result.

When first developing your application, it can be helpful to display the literal within a message box using OS.Alert() in order to review the contents that are returned in your environment. You can then tailor your JavaScript to process the contents.

```
var getListResult = OS.File.GetList( "/System/*" );
OS.Alert("GetList results:\n" + getListResult );
```

Following is a sample of one technique to test for a property before using the property. This technique uses the JavaScript Object.hasOwnProperty() method to check for existence. The File.GetList() method may return a result which is missing the lastAccessTime for directory entries

```
// Sample GetList literal result
[ {name:"myApp.cab", attributes:0x21,
 creationTime:new Date(2006,11,15,11,51,41,480),
 lastAccessTime:new Date(2007,7,27,3,27,41,0),
 lastWriteTime:new Date(2008,6,15,0,29,50,0), size:2455494},
{name:"myconfig.ini", attributes:0x21,
 lastAccessTime:new Date(2007,7,27,3,27,18,0),
 lastWriteTime:new Date(2008,6,15,0,29,48,0), size:12564},
{name:"AppDirectory", attributes:0x10,
 lastWriteTime:new Date(2008,2,11,12,29,49,0), size:1024} ]


var getListResult = OS.File.GetList( "/System/*" );
eval("var fileArray=" + getListResult );
var i;
var file;
var time;
for (i=0; i<fileArray.length; ++i)
{
    file = fileArray[i];
    if (file.hasOwnProperty( "lastAccessTime" ))
    {
        time = file.lastAccessTime;
        OS.Alert( "Last Access in " + time.getFullYear() );
    }
}
```

3.0 CETerm Automation Objects

This section describes the Automation Objects available to the CETerm script engine. These objects provide access to the running CETerm application, TE session screens, the Windows CE operating system, hardware device components, and other features for developing rich applications.

The automation objects are accessed in a hierarchical manner similar to the Document Object Model (DOM) of a webpage. The three top-level objects are `CETerm, Device,` and `OS`. The `CETerm` object provides access to application specific features. The `Device` object provides access to device hardware such as keyboards and serial ports. The `OS` object provides access to generic Operating System (OS) features such as files, events, processes, the network, and the Windows registry.

Automation objects provide some of the same functionality provided by the Window object in the web browser. For example, the familiar Window methods `alert()` and `setTimeout()` are provided by the `OS.Alert()` and `CETerm.SetTimeout()` methods.

Many of the automation objects on the OS hierarchy give direct access to low-level Windows CE features. Although powerful, caution should be exercised when using these features. We provide the basic API documentation in this document. More information and useful discussion can be found by searching the Microsoft msdn.microsoft.com website. Below, we suggest such searches and provide relevant keywords.

The top-level CETerm objects are described in the first 3 sections below, followed by all other objects in alphabetical order.

3.1 THE CETERM OBJECT

The top-level `CETerm` object gives access to CETerm features, settings, and session state. This section documents the methods and properties of the `CETerm` object.

Methods

The following methods are available

Method	**Action**
AbortScript	Abort the currently running script
ClearAllTimers	Clear all SetTimeout and SetInterval timers
ClearInterval	Clear a recurring interval timer

ClearTimeout	Clear a one-time timer
GetProperty	Get a property value
PlaySound	Play a tone or wave file on the device (deprecated)
PlayTone	Play a tone on the device (deprecated)
PostIDA	Send a command to a session (asynchronous)
RunScript	Run a script (called from a web browser only)
SendIDA	Send a command to a session (synchronous)
SendText	Send text to a session
Session	Get a session object
SetInterval	Create a recurring interval script execution timer
SetProperty	Set a property value
SetScriptTimeout	Set the current script execution timeout
SetTimeout	Create a one-time script execution timer

AbortScript ( )

Stop the currently executing script.

ClearAllTimers ( )

Clear all recurring interval timers and one-time timers.

ClearInterval (intervalTimerID)

Clear the specified recurring interval timer.

ClearTimeout (timerID)

Clear the specified one-time timer.

value = GetProperty (propertyName)

Return the named property value. This may be a device property, application property, or session property. See Appendix 2 for a list of available properties. Returns the JavaScript “undefined” value if the requested property cannot be found.

PlaySound (sound)   (deprecated)

Play a tone or wave file on the device. This PlaySound is not the same as the Windows PlaySound of the OS object. This method will accept a wave file name but it will also accept a “tone specifier” as a string to support this legacy feature in the Naurtech Web Browser. New application should use the OS.PlaySound or OS.PlayTone commands. Use the complete file path if the wave file is not in the \Windows directory.

If the handheld device contains a programmable tone generator, the sound parameter may also be a string which defines a sequence of tones to play. The syntax is given below:

“vvfffddd” – where
vv – is the volume 01-10
fff – is the frequency in 10’s of MHz, 000-999
ddd – is the duration in 10’s of milliseconds, 000-999

Multiple tone specifications can be concatenated.

PlayTone(volume, frequency, duration) (deprecated)

Play a tone if supported by the handheld hardware. This method is provided for backward compatibility within the web browser. New application should use OS.Playtone() which provides the same functionality.

volume – is the volume 00 -10 (0 is off, 10 is loudest)
frequency – is the frequency in Hz.
duration – is the duration in milliseconds.

PostIDA (IDASymbolicName, session)

PostIDA submits an IDA action command and directs it to the specified session. Valid session values are 1 to MaxSession. The special session value of 0 will send the command to the current session. Some IDA commands act at a global level and ignore the session variable. See Appendix 1 for IDA Symbolic Names.

The PostIDA command will return before the action executes. In general, the IDA action will not be applied until after the current script execution ends. We recommend using PostIDA rather than SendIDA. There are only rare situations when SendIDA must be used.

status = RunScript (script)

Run the specified script in the CETerm engine. This method must only be used when the CETerm object is referenced from the web browser script engine. In general, it is better to use PostIDA with an IDA_SCRIPT_xx action to run a pre-defined script from the web browser. To execute a script contained in a string from the CETerm engine use the JavaScript "eval()" method.

SendIDA (IDASymbolicName, session)

SendIDA sends an IDA action command and directs it to the specified session. Valid session values are 1 to MaxSession. The special session value of 0 will send the command to the current session. See Appendix 1 for IDA Symbolic Names.

The SendIDA method will attempt to complete the action before returning. We recommend using PostIDA rather than SendIDA. There are only rare situations when SendIDA must be used. For example, SendIDA will be needed if you need to invoke IBM field actions, such as Field Exit, between sending text to an IBM session with SendText.

SendText (text, session)

SendText sends a text string to the specified terminal emulation session. Valid session values are 1 to MaxSession. The special session value of 0 will send the command to the current session. This command is synchronous and CETerm will act on each character before this method returns. SendText will not send text to a browser session. To change text into a browser page, use the `Browser.Document` reference and assign the text directly to the desired page element.

The text string may include IDA symbolic names between backslash characters '\'. The IDA codes will be interpolated as the text is sent. For example, "username\\IDA_FIELD_EXIT\\secretpassword". Note that each backslash has a preceding backslash because it is the JavaScript "escape" character. To put a single literal backslash in a string you precede it with another backslash.

object = Session (index)

Return the corresponding Session object. Valid index values are 1 to MaxSession. The object is returned even if the session is not connected.

intervalTimerID = SetInterval (scriptExpression, delayMillisec)

Set a recurring interval timer to execute the scriptExpression after each delay of delayMillisec. This method returns an ID that should be saved in a global variable for later use with ClearInterval if needed. Other scripts may run while waiting for this timer. The scriptExpression is a string containing the script, but is commonly a function invocation, such as “myTimerFunction(3, ‘alert’);”

Timers are especially useful with complex or long-running scripts. Interval timers can be used to perform simple update tasks. One-time timers should be used in preference to interval timers. In general, scripts should perform a short action and exit. With a complex script such as a state-machine, the state can be maintained in global variables and the script re-activated periodically to check for state transitions and perform actions. See the “expect” script and “ExpectMonitor” class in Section 5.1 for an example of the use of a timer.

status = SetProperty (propertyName, propertyValue)

SetProperty will assign the given value to the named property. See Appendix 2 for a list of available properties. The returned status is 0 for success, non-zero for failure.

SetScriptTimeout (millisec)

Set the maximum script execution time. This value may be updated during a running script. If updated, the new timeout will apply starting at the time of the change. A value of 0 will disable the timeout.

The script timeout prevents a faulty script from locking-up CETerm. For example, if a script enters an “endless loop”, the timeout will eventually force the script to abort.

timeoutTimerID = SetTimeout (scriptExpression, delayMillisec)

Set a one-time timer to execute the scriptExpression after a delay of delayMillisec. This method returns an ID that should be saved in a global variable for later use with ClearTimeout if needed. Other scripts may run while waiting for this timer. The scriptExpression is a string containing the script, but is commonly a function invocation, such as “myTimerFunction(3, ‘alert’);”

Timers are especially useful with complex or long-running scripts. Timers can also be used to defer an operation which is not possible within an event handler. One-time timers should be used in preference to interval timers. In general, scripts should perform a short action and exit. With a complex script such as a state-machine, the state can be maintained in global variables and the script re-activated periodically to check for state transitions and perform actions. See the “expect” script and “ExpectMonitor” in Section 5.1 for an example of the use of a timer.

Properties

The `CETerm` object has the following properties.

Property	Description	Values
ActiveSession	Current active session. (read only)	1-MaxSession
MaxSession	Maximum session index. (read only)	5
Message	Returns message object. (read only)	object
TextInput	Return text input object. (read only)	object

3.2 THE DEVICE OBJECT

The top-level `Device` object provides access to device components such as the keyboard and serial ports. This section documents the methods and properties of the `Device` object.

Not all features of the `Device` object will be available on all devices. For example, GPS, RFID, Speech, and Trigger functionality will depend on the hardware make, model, and operating system version.

Methods

The following methods are available

Method	Action
GetBatteryInfo	Get the battery charge information.
GetPowerState	Get the current power state for a device component.
PowerStateRequest	Request a change of the power state of a device component.
ResetIdleTimer	Reset the Windows idle timer to prevent a suspend.
SerialPort	Return the requested SerialPort object.
Vibrate	Activate the vibrator.

batteryStatus = GetBatteryInfo ( )

Get the battery charge information. Returns a JavaScript object literal with the information in the form:

```
{ACLineStatus:0x1,
 main:{flags:0x1, lifePercent:100, lifeTime:4294967295,
    fullLifeTime:4294967295, voltage:4178, current:0,
    averageCurrent:0, averageInterval:0,
    mAHourConsumed:0, temperature:3.8, chemistry:0x4},
 backup:{flags:0x1, lifePercent:100, lifeTime:4294967295,
    fullLifeTime:4294967295, voltage:2797}}
```

See Section 2.8 for details about handling JavaScript literals. See Appendix 4 for status and flag definitions. Times are in seconds, voltages are in millivolts, and currents are in milliamperes. Depending on the device, some values may be invalid. When a battery is under charge (ACLineStatus:0x1) some values may be invalid. Return null if battery information is not available. Search msdn.microsoft.com with keyword “SYSTEM_POWER_STATUS_EX2” for more details.

status = GetPowerState (deviceName)

Get the power state of the specified device component. The valid device names depend on the hardware. Common devices are serial ports (“COMx:”) and the backlight (“BKL1:”). Return values are 0 – full on, 1 – low on, 2 – standby, 3 – sleep, 4 – off, or -1 if unknown. Check the `Device` property LastError for more error information.

status = PowerStateRequest (deviceName, powerState)

Request the OS to set the power state of a device component. The state is one of the 5 values listed under GetPowerState. See Appendix 4 for state constants. Return 0 for success, non-zero for failure. Check the `Device` property LastError for more error information.

ResetIdleTimer ( )

Resets the system idle timer. The idle timer is used to determine when to enter a suspended state. Resetting the timer will typically prevent the device from suspending.

object = SerialPort (index)

Return the corresponding SerialPort object. Valid index values are 0 through 9. See Chapter 5 for details on how to use the SerialPort object.

status = Vibrate(durationMillisec)

Activate the device vibrator for the specified duration in milliseconds. Return 0 on success or -1 otherwise. For most devices, this method is synchronous and will not return until the vibration is complete.

Properties

The `Device` object has the following properties.

Property	Description	Values
GPS	Returns the `GPS` object. This object provides access to GPS actions. (read only)	object
Keyboard	Returns the `Keyboard` object. This object provides access to keyboard actions. (read only)	object
RFID	Returns the `RFID` object. This object provides access to the RFID device. The RFID object is available only in select builds of CETerm and is documented separately. (read only)	object
Speech	Returns the `Speech` object. This object provides access to the Speech features. The Speech object is available only in select builds of CETerm and is documented separately. (read only)	object
Trigger	Returns the `Trigger` object. This object provides access to hardware trigger features. The Trigger object is available only for limited devices and is documented separately. (read only)	object
LastError	Returns the last Windows error related to the Device object. (read only)	unsigned integer

3.3 THE OS OBJECT

The top level `OS` object provides access to operating system resources such as files, processes, windows, and the registry.

Many of the automation objects on the OS hierarchy give direct access to low-level Windows CE features. Although powerful, caution should be exercised when using these features. We provide the basic API documentation in this document. More information and useful discussion can be found by searching the Microsoft msdn.microsoft.com website. Below, we suggest such searches and provide relevant keywords.

Methods

The following methods are available

Method	Action
Alert	Show the user a text message. (synchronous)
Beep	Play a default beep tone.
Exec	Run a separate program. (deprecated, use Process object)
GetErrorMessage	Get a text error message for a Windows CE error value.
KillProcess	Stop a running process started with Exec. (deprecated)
MessageBox	Display a standard Windows MessageBox.
PlaySound	Play a wave file on the device.
PlayTone	Play a tone on the device.
Sleep	Pause the script execution.
StopSound	Stop an asynchronous playing PlaySound sound.
WaitForProcess	Wait for the specified process to end. (deprecated)

Alert (message)

Show the user a simple text message and wait for them to press OK.

Beep ( )

Sound the default Windows beep tone.

status = Exec(programFile, commandLine)

Start the specified program. Return 0 for success, non-zero for failure.

Use GetErrorMessage() to convert a non-zero status to a text message. You should immediately save the property LastExecProcess after a successful Exec call to obtain the process ID for later use in WaitForProcess or KillProcess. The programFile should be a fully qualified filename. **This function is deprecated**. Full process control is now provided by the `Process` object.

text = GetErrorMessage (error)

Return a descriptive text message for the specified Windows error. If the value is unknown, return the error hexadecimal value as text.

status = KillProcess(processID)

Attempts to stop the specified process. Returns 0 for success, non-zero for failure. You must obtain the processID from the property LastExecProcess immediately after a successful Exec call. **This function is deprecated**. Full process control is now provided by the `Process` object.

result = MessageBox (message, title, flags)

Display a standard Windows message box. The title is displayed in the message box title bar. The flags are used to specify the icon and buttons that are visible. Return a value corresponding to the button pushed to close the dialog. See Appendix 4 for flag definitions. Search msdn.microsoft.com with keywords “messagebox ce” for more details.

PlaySound (sound, flags)

Play a wave file on the device. This PlaySound is not the same as the CETerm.PlaySound(). Use the complete absolute file path if the wave file is not in the \Windows directory. The flags control the way the sound is played. See Appendix 4 for flag definitions. Returns true on success, false otherwise. Search msdn.microsoft.com with keywords “playsound ce” for more details.

PlayTone (volume, frequency, duration)

Play a tone if supported by the handheld hardware. New applications should use this method and avoid CETerm.Playtone().

volume – is the volume 00 -10 (0 is off, 10 is loudest)
frequency – is the frequency in Hz.
duration – is the duration in milliseconds.

Sleep (delay)

Delay script execution for specified milliseconds.

result = StopSound ( )

Stop any currently playing sound. Returns 0 on success.

status = WaitForProcess(processID, timeout)

Wait for the specified process to exit. Return after timeout milliseconds even if process is still running. Return 0 if process has exited, non-zero for timeout or failure. You must obtain the processID from the property LastExecProcess immediately after a successful Exec call. **This function is deprecated**. Full process control is now provided by the Process object.

Properties

The `OS` object has the following properties.

Property	Description	Values
ClipboardData	Provides access to the current Windows clipboard “cut and paste” buffer. Assigning to this property will set the clipboard contents.	string
Event	Returns the Event object. This object provides access to the Windows events used for synchronization. (read only)	object
File	Returns the File object. This object provides access to the Windows file systems. (read only)	object
LastError	Returns the last Windows error related to the OS object.	integer
LastExecProcess	Returns the process ID of the last program started via Exec. (read only) (deprecated)	unsigned integer
LastOSError	Returns the last Windows OS error. (read only).	integer
MemoryStatus	Returns a summary of the Windows memory available. (read only)	object literal string

Property	Description	Values
Network	Returns the Network object. This object provides access to Windows network features. (read only)	object
Process	Returns the Process object. This object provides control of running Windows programs. (read only)	object
Registry	Returns the Registry object. This object provides access to the Windows registry. (read only)	object
TickCount	Returns the current tick count from Windows. This provides a millisecond resolution time source. (read only)	unsigned integer, 0-0xFFFFFFFF
Window	Returns the Window object. This object provides access to current windows of running programs. (read only)	object

MemoryStatus

The return value is in the form of a JavaScript object literal. See Section 2.8 for information on handling literal return values.

For example the following object literal shows results from a Windows CE 5.0 device:

```
{utilization:22, totalRAM:58613760, availableRAM:46153728,
 totalStorage:29061120, availableStorage:17583852}
```

Not all devices will return all values, so you should check for the existence of a value before use. See Section 2.8 for details.

TickCount

The return value is the number of milliseconds since the device booted, excluding any time that the system was suspended. TickCount starts at zero on boot and then counts up from there. The count will rollover to zero if the system is run continuously for 49.7 days. The maximum value is 0xFFFFFFFF.

When using TickCount, beware that rollover may occur. Comparing tick values directly does not always yield the correct results. By design, TickCount may have a drift of 1 second per 2 hours. Do not use TickCount for drift sensitive applications.

3.4 THE BROWSER OBJECT

The `Browser` object gives access to a web browser session. The `Browser` object is a property of the `Session` object; `CETerm.Session(i).Browser`. This section documents the methods and properties of the `Browser` object.

Methods

The following methods are available

Method	Action
AddMetaItem	Add a CETerm <META> element to the current web page.
Navigate	Navigate to specified URL.
RunScript	Run a script in the web browser JavaScript engine.

result = AddMetaItem (target, content)

Add a CETerm <META> tag element to the current web page. This is typically used to add custom <META> elements which define key mappings or other custom behaviors. See the Naurtech Web Browser Programming Guide for documentation on custom <META> tags. Return 0 for success, non-zero for failure. After adding META elements that change the values of information icons you may need to use CETerm.PostIDA(“IDA_INFO_REFRESH”, 0) to apply the changes.

result = Navigate (URL)

Navigate the browser session to the specified URL. Return 0 for success, non-zero for failure.

result = RunScript (script)

Execute the specified script in the browser JavaScript engine. Return 0 for success, non-zero for failure.

Properties

The following read-only properties are available.

Property	Description	Values
Document	Document object of the current web page. The DOM of the page may be examined and altered via this object. WARNING: Use a local variable to hold this reference to minimize memory usage. (read only)	object
DocLoaded	Returns true if document is loaded. (read only)	true, false

3.5 THE EVENT OBJECT

The `Event` object provides access to the Windows "event objects". The `Event` object is a property of the `OS` object; `OS.Event`. Windows event objects are used to synchronize operations between processes and signal special conditions. Normally, Windows event objects are used within a single program or between programs designed to work together. By providing access to event objects through scripting, CETerm makes a rich environment to control and interact with separate applications. For example, a custom utility program written to control a special device peripheral can signal an event to inform CETerm that it should read data and respond to a host.

Windows event objects should only be used when synchronization with external programs or device services is required. For a better understanding of Windows event objects, you can search for information at msdn.microsoft.com with the keywords "createevent ce".

Methods

The following methods are available

Method	Action
ClearAllListeners	Clear all assigned handler scripts.
ClearListener	Clear an assigned handler script for a single event.
Create	Create a handle for a named event. If the named event does not yet exist within Windows, it is created.
Delete	Delete an event handle.
DeleteAllEvents	Delete all event handles open in CETerm.

GetHandlerScript	Returns the handler script for an active listener.
IsExistingEvent	Returns status of an event handle.
IsListenerSet	Returns status of a listener on a single event.
Pulse	Signals all listeners that an event is set, then resets the event to the non-signaled state
Reset	Reset an event to the non-signaled state.
Set	Set an event to the signaled state.
SetListener	Assign a handler script to an event. The handler is run when the event is signaled.
SetProcessListener	Assign a handler script to a running process. The handler is run when the process exits.

status = ClearAllListeners ( )

Clear all handler scripts that have been assigned to events. ClearAllListeners will also clear handler scripts waiting for processes. Return 0 for success, or a negative value for failure.

status = ClearListener (event)

Clear the handler script for the specified event. The event may be specified by the integer event handle or the event name. Return 0 for success, or a negative value for failure.

eventHandle = Create (eventName, manualReset)

Create an eventHandle for the named event. The eventName cannot be empty. If manualReset is true, the event will not be reset after waiting for a listener. If manualReset is false, the event is automatically reset after waiting for a listener. If the named event already exists within Windows, the manualReset value is ignored and a handle to the existing event is returned. Otherwise, the event is created within Windows.

Create is typically used when a new event is created for use exclusively within CETerm. Create may also be used to create a handle within CETerm to access an event which is normally created within Windows by another program or driver.

Return the eventHandle for success, 0 for failure.
Use the `Event` property LastError to get additional error information.

status = Delete (eventHandle)

Delete the specified eventHandle from management. The Create() method must have been used to create the eventHandle. For convenience, you may specify the eventName used in the Create() call rather than the eventHandle. Delete() will clear the event listener if it exists. After deleting an eventHandle, it can no longer be used for any event operations. If Delete() closes the last open handle to the Windows event, the named event will no longer exist within Windows.

Return 0 for success, or a negative value for failure. Use the `Event` property LastError to get additional error information.

status = DeleteAllEvents ( )

Delete all eventHandles obtained with Create(). All event listeners associated with the events are cleared. DeleteAllEvents() will not clear event listeners that have been assigned, by name, to Windows events created by other programs.

Return 0 if any events are deleted, or a negative value if none deleted.

script = GetHandlerScript (eventHandleOrName)

Return the handler script associated with the specified eventHandle or eventName. Return null if no listener found.

script = GetList ( )

Return a list of events from Create() calls and active SetListener() handlers. The returned list is in the form of a JavaScript array literal […] which contains JavaScript object literals {…} containing information about each event. See Section 2.8 for details about handling JavaScript literals. Return empty array or null if no events found.

Names of events are included in the results if specified in a Create() call. The manualReset is included if the Create() call actually created the corresponding Windows CE event.

Here is a sample event list output:

```
[{id:0xFE490034,name:"MyPrivateEvent",manualReset:false},
 {id:0xF345DE00,name:"ExistingEvent"},{id:0xEF546902}]
```

script = GetName (eventHandle)

Return the name of the event associated with the specified eventHandle. Return null if no listener found.

status = IsExistingEvent (eventHandleOrName)

Return the status of the event with the given eventHandle or eventName. Return 0 if the event does not exist, 1 if the eventName exists within Windows, 2 if the event was created within Windows by the Create() method, or -1 on error.

status = IsListenerSet (eventHandleOrName)

Return the status of the listener with the given eventHandle or eventName. Return 0 if a listener is not set, 1 if a listener is set, or -1 on error.

status = Pulse (eventHandleOrName)

Pulse the state of an event. This signals all listeners that an event is set, then resets the event to the non-signaled state. The eventHandle is obtained from a Create() call. If you use an eventName, the event must have been previously created within Windows by Create() or by another program.

Return 0 on success or -1 if no matching event to pulse.

status = Reset (eventHandleOrName)

Reset an event to the non-signaled state. The eventHandle is obtained from a Create() call. If you use an eventName, the event must have been previously created within Windows by Create() or by another program.

Returns 0 on success or -1 if no matching event to reset.

status = Set (eventHandleOrName)

Set an event to the signaled state. The eventHandle is obtained from a Create() call. If you use an eventName, the event must have been previously created within Windows by Create() or by another program.

Return 0 on success or -1 if no matching event to set.

status = SetListener (eventHandleOrName, handlerScript, timeout)

Assign a handler script to an event. The eventHandle is obtained from a Create() call. If you use an eventName, the event must have been previously created within Windows by Create() or by another program. The handlerScript will be queued for execution if the event is signaled within timeout milliseconds. The handler script is not invoked if the listener times out.

The special timeout value of 0xFFFFFFFF will never timeout. The special timeout value 0 will cause an immediate check of the event state. NOTE: The currently running script which invoked SetListener must finish before any handler can be executed.

Return 0 if a listener is set or a negative value on error.

status = SetProcessListener (processID, handlerScript, timeout)

The SetProcessListener method assigns a handler to the special event that occurs when a process exits. The processID is the process id number assigned when the process is created by Windows. You can find process id values using the `Process` object methods. The handlerScript will be queued for execution if the process exits within timeout milliseconds. The handler script is not invoked if the listener times out.

The special timeout value of 0xFFFFFFFF will never timeout. The special timeout value 0 will cause an immediate check of the event state. NOTE: The currently running script which invoked SetProcessListener must finish before any handler can be executed.

Return 0 if a listener is set or a negative value on error.

Properties

The `Event` object has the following properties.

Property	Description	Values
LastError	Returns the last error value associated with the Event object.	unsigned integer

3.6 THE FILE OBJECT

The `File` object provides access to the Windows file system. The `File` object is a property of the `OS` object; `OS.File`.

Methods

The following methods are available

Method	Action
Append	Append content to a file.
Copy	Create a copy of an existing file.
CreateDirectory	Create a new directory.
Delete	Delete an existing file.
GetAttributes	Get the attributes of an existing file.
GetList	Get list of files with names that match a pattern.
GetOpenFileName	Select a filename with a file Open dialog.
GetSaveFileName	Select a filename with a file Save dialog
Move	Move or rename a file.
Read	Read file contents.
RemoveDirectory	Remove (delete) an existing directory.
SetAttributes	Set the attributes of an existing file.
Write	Write contents to a new or existing file.

status = Append (fileName, content)

Append content to the file. The content is specified as a text string. Return true for success, false for failure. If the file does not exist, it is created. Use the `File` properties LastError or LastErrorMessge to get additional error information.

status = Copy (existingFile, newFile, overWrite)

Copy an existing file to a new file. If a file already exists with the new file name, copy will fail unless overWrite is true. Return true for success, false for failure. Use the `File` properties LastError or LastErrorMessge to get additional error information.

status = CreateDirectory (newDirectory)

Create a new directory. Return true for success, false for failure.
Use the `File` properties LastError or LastErrorMessge to get additional error information.

status = Delete (fileName)

Delete an existing file. Return true for success, false for failure.
Use the `File` properties LastError or LastErrorMessge to get additional error information.

attributes = GetAttributes (fileName)

Return the attributes of the file. Use the `File` properties LastError or LastErrorMessge to get additional error information. See Appendix 4 for attribute definitions.

list = GetList (pattern)

Return a list of files with names that match a pattern. The pattern specifies a valid directory or path and file name, which can contain wildcard characters, such as an asterisk (*) or a question mark (?). You may use forward slashes (/) to delimit directories. The returned list is in the form of a JavaScript array literal […] which contains JavaScript object literals {…} containing information about every matching file. See Section 2.8 for details about handling JavaScript literals. Return null if no matching files found. Use the `File` properties LastError or LastErrorMessge to get additional error information.

For example the following array literal shows two files and a directory:

```
[ {name:"myApp.cab", attributes:0x21,
 creationTime:new Date(2006,11,15,11,51,41,480),
 lastAccessTime:new Date(2007,7,27,3,27,41,0),
 lastWriteTime:new Date(2008,6,15,0,29,50,0), size:2455494},
{name:"myconfig.ini", attributes:0x21,
 lastAccessTime:new Date(2007,7,27,3,27,18,0),
 lastWriteTime:new Date(2008,6,15,0,29,48,0), size:12564},
{name:"AppDirectory", attributes:0x10,
 lastWriteTime:new Date(2008,2,11,12,29,49,0), size:1024} ]
```

Note that the “creationTime” and “lastAccessTime” are not always present and should be checked for existence before using them. Their existence depends on the type of Windows filesystem holding the files. See any standard JavaScript reference for details on the Date() constructor arguments. See Appendix 4 for attribute definitions.

filename = GetOpenFileName (title, filter)

Return the name of a file specified by the user in an Open file dialog. The title of the dialog should contain descriptive information for the user. For example, “Please select a datafile.” The filter is a list of filter pairs. Each pair represents the description of a filter and the file selector wildcards. For a JavaScript file it may look like this: `"JScript File (*.js)\x00*.js\x00\x00"`. Each element of the pair is followed by `"\x00"` as a separator character. The last pair has an additional trailing `"\x00"`. Here is a multiple filter example:

```
   "JScript File (*.js)\x00*.js\x00All Files(*.*)\x00*.*\x00\x00"
```

filename = GetSaveFileName (title, filter)

Return the name of a file specified by the user in a Save file dialog. The title of the dialog should contain descriptive information for the user. For example, “Save file as:”. The filter is list of filter pairs. Each pair represents the description of a filter and the file selector wildcards. For a JavaScript file it may look like this: `"JScript File (*.js)\x00*.js\x00\x00"`. Each element of the pair is followed by `"\x00"` as a separator character. The last pair has an additional trailing `"\x00"`. Here is a multiple filter example:

```
   "JScript File (*.js)\x00*.js\x00All Files(*.*)\x00*.*\x00\x00"
```

status = Move (existingFilename, newFileName)

Move or rename an existing file. Returns true for success, false for failure. Use the `File` properties LastError or LastErrorMessge to get additional error information.

contents = Read (fileName)

Read entire file and return as contents. The read is an atomic operation which opens the file, reads all contents and closes the file. The File object does not support the concept of an “open” file or reading parts of a file. There must be sufficient memory to hold the entire file contents. There is no error information returned. Use GetAttributes to validate a filename and ensure read access.

status = RemoveDirectory (directoryName)

Delete an existing directory. Return true for success, false for failure.

Use the `File` properties LastError or LastErrorMessge to get additional error information.

status = SetAttributes (fileName, attributes)

Set the attributes of the file. Return true for success, false for failure. Use the `File` properties LastError or LastErrorMessge to get additional error information. See Appendix 4 for attribute definitions.

status = Write (fileName, contents)

Write contents to the named file. Return true for success, false for failure. Any current contents are first deleted. The write is an atomic operation which opens the file, writes all contents and closes the file. The `File` object does not support the concept of an "open" file or writing parts of a file. To append to a file, use Append. Use the `File` properties LastError or LastErrorMessge to get additional error information.

Properties

The `File` object has the following properties.

Property	Description	Values
LastError	Returns the last error value associated with the File object.	unsigned integer
LastErrorMessage	Returns a text message of the last error associated with the File object. (read only)	text

3.7 THE FTP OBJECT

The `FTP` object provides access to the File Transfer Protocol (FTP) operations.

You must use the Login() method prior to using the other methods. The `FTP` object is a property of the `Network` object; `OS.Network.FTP`.

Methods

The following methods are available

Method	Action
CreateDirectory	Create a new directory on the remote host.
DeleteFile	Delete a file on the remote host.
DeleteDirectory	Delete a directory on the remote host.
GetFile	Get a file from the remote host.
GetDirectory	Get the current directory on the remote host.
ListFiles	List files on the remote host.
Login	Login to the FTP service of the remote host.
Logout	Logout of the FTP service.
PutFile	Put a local file onto the remote host.
RenameFile	Rename a file on the remote host.
SetDirectory	Set the current directory on the remote host.

status = CreateDirectory (directoryName)

Create a new directory on the remote host. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = DeleteFile (fileName)

Delete the named file on the remote host. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = DeleteDirectory (directoryName)

Delete the named directory on the remote host. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = GetFile (localName, remoteName)

Copy the named remote file to the given local name. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

directory = GetDirectory ( )

Return the name of the current working directory on the host. Returns null on failure. Use the `FTP` properties LastError and LastErrorText to get additional error information.

filelist = ListFiles (pattern)

Return a list of remote files with names that match a pattern. The pattern specifies a valid directory path or file name on the remote host, which can contain wildcard characters, such as an asterisk (*) or a question mark (?); but may not contain spaces. An empty pattern will list all files in the current working directory. The returned list is in the form of a JavaScript array literal […] which contains JavaScript object literals {…} containing information about every matching file. See Section 2.8 for details about handling JavaScript literals. Return an empty array literal if no matching files are found. Use the `FTP` properties LastError or LastErrorText to get additional error information.

The returned object literals will depend on the remote host and may not contain all possible properties. See Section 2.8 for dealing with missing properties. For example the following array literal shows two files and a directory:

```
[ {name:"myApp.cab", attributes:0x21,
 creationTime:new Date(2006,11,15,11,51,41,480),
 lastAccessTime:new Date(2007,7,27,3,27,41,0),
 lastWriteTime:new Date(2008,6,15,0,29,50,0), size:2455494},
{name:"myconfig.ini", attributes:0x21,
 lastAccessTime:new Date(2007,7,27,3,27,18,0),
 lastWriteTime:new Date(2008,6,15,0,29,48,0), size:12564},
{name:"AppDirectory", attributes:0x10,
 lastWriteTime:new Date(2008,2,11,12,29,49,0), size:1024} ]
```

Note that the “creationTime” and “lastAccessTime” are not always present and should be checked for existence before using them. Their existence depends on the type of remote host. See any standard JavaScript reference for details on the Date() constructor arguments. See Appendix 4 for attribute definitions.

status = Login (hostname, userName, password)

Establish an FTP connection to the remote named host. The userName and password are used for authentication. If the default values are not correct, you must configure the Port and PassiveMode properties before Login. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = Logout ( )

Close an open FTP connection. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = PutFile (localName, remoteName)

Copy the named local file to the remote name. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = RenameFile (existingName, newName)

Rename an existing remote file to the newName. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

status = SetDirectory (newWorkingDirectory)

Set the current working directory on the remote host to the specified directory. Return 0 on success or non-zero otherwise. Use the `FTP` properties LastError and LastErrorText to get additional error information.

Properties

The `FTP` object has the following properties.

Property	Description	Values
ASCIIMode	Use ASCII mode for file transfers if true. Use binary mode if false. ASCII mode may change the line-termination characters in files. This setting may be changed while a session is connected. Default is false.	true or false
HostName	Hostname of the current active session. (read only)	string

Property	**Description**	**Values**
LastError	Returns the last error value associated with any FTP operation. (read only)	unsigned integer
LastErrorText	Returns a text description of the last FTP error as reported by the remote host. (read only)	string
LoggedIn	True if FTP session is currently established. (read only)	true or false
OverwriteExistingLocalFile	If true, GetFile() will overwrite an existing file with the same local name. If false, GetFile() will report an error if a file with the same name exists. Default is false.	true or false
PassiveMode	If true, use passive FTP semantics for the connection. This must be set prior to login. Default is false.	true or false
Port	Specifies the TCP/IP port used on the server for the FTP connection. Default is 21.	unsigned integer
ServerListsUTCFiletimes	If true, ListFiles() assumes that the server sends UTC based file timestamps and converts these to local times before output. If false, assumes that timestamps are in local times. Default is false.	true or false
UserName	User name of the current active session. (read only)	string

3.8 THE GPS OBJECT

The `GPS` object provides access to GPS operations. The `GPS` object is a property of the `Device` object; `Device.GPS.` The `GPS` object provides direct access to the GPS functionality. Some devices do not support the `GPS` object, but GPS data can be read from a serial port using the `Device.SerialPort()` object. Consult your device documentation for details. Supported for CETerm V5.5.3 or later.

Methods

The following methods are available

Method	Action
Open	Open the GPS device and enable operations.
Close	Close the GPS device.
GetPosition	Read the current GPS position.
GetDeviceState	Query the GPS device state.

status = Open ( )

Open the GPS device and enable operations. This action will supply power to the GPS device if it is not already operational. To minimize power consumption, the GPS device should remain closed until data is required.

Before opening the device, you may want to set the event properties if events are used to monitor location changes. Return 0 for success, non-zero for error. Use the `GPS` property LastError to get additional error information.

status = Close ( )

Close the GPS device. Return 0 for success, non-zero for error. Use the `GPS` property LastError to get additional error information.

position = GetPosition( )

Read the current GPS position. The returned position is a JavaScript object literal {…} containing information about the current position. See Section 2.8 for details about handling JavaScript literals. Return null if error. Use the `GPS` property LastError to get additional error information.

The returned literal contents will depend on the capabilities of the GPS hardware. If successful, the contents should at least contain “latitude” and “longitude” values. The `GPS` property MaximumAge also affects the returned values. Any position data older than MaximumAge milliseconds will not be reported. Satellite data is only included if the `GPS` property IncludeSatelliteData is true.

Here is a sample position output:

```
{flags:0x0,timeUTC:{year:2008,month:8,day:8,
 hour:8,minute:8,second:0,millisecond:0},
 latitude:47.64000000,longitude:-122.13000000,
 speed:0.0,heading:0.0,magneticVariation:0.0,
 altitudeSeaLevel:30.123,altitudeEllipsoid:50.00,
 fixQuality:1,fixType:1,fixSelection:1,
 positionalDOP:0.0,horizontalDOP:0.0,verticalDOP:0.0,
 satellites:{usedCount:4,usedPRN:[0,0,0,0],
 inViewCount:8,inViewPRN:[0,0,0,0,0,0,0,0],
 inViewElevation:[0,0,0,0,0,0,0,0],
 inViewAzimuth:[0,0,0,0,0,0,0,0],
 inViewSNR:[0,0,0,0,0,0,0,0]} }
```

The speed is in knots and altitudes are in meters.

If GetPostion() is called when the `GPS` object is closed, it may return a cached position reading from the device, but will not apply power. If the reading was cached but is older than MaximumAge, it will not be returned. You should check timeUTC to determine if the returned values were cached.

state = GetDeviceState( )

Read the GPS device state. The returned state is a JavaScript object literal {…} containing information about the current device state. See Section 2.8 for details about handling JavaScript literals. Return null if error. Use the `GPS` property LastError to get additional error information.

The returned literal contents will depend on the GPS hardware. Here is a sample state output:

```
   {serviceState:0x1,deviceState:0x0,
    timeLastDataReceived:new Date(2008,8,8,8,8,0),
    driverPrefix:"COM4:",multiplexPrefix:"GPD1:",
    friendlyName:"ACME GPS Card, version 1.23"}
```

Properties

The `GPS` object has the following properties.

Property	Description	Values

Property	Description	Values
DeviceStateChangeEvent	Event ID of event signaled when GPS device state changes. If needed, you must set this value before calling Open(). The eventID must be obtained by calling OS.Event.Create()	eventID
IncludeSatelliteData	Include satellite data with position results if true. Default is false.	true, false
IsOpen	Indicates GPS device is open and enabled. (read only)	true, false
LastError	Returns the last error value associated with the `GPS` object.	unsigned integer
LastPosition	Return last position data obtained by GetPosition.	object literal
MaximumAge	Maximum age in milliseconds of position results returned by GetPosition. Default value is 180000.	unsigned integer
NewLocationDataEvent	Event ID of event signaled when GPS position changes. If needed, you must set this value before calling Open(). The eventID must be obtained by calling OS.Event.Create().	eventID

Example

The following example shows how the GPS device can be monitored and the current location displayed. This example also uses the `OS.Event` methods, a global hot-key, and `CETerm.Message`. The event is signaled by the GPS device and used to update the display. The hot-key is used to terminate the demo. To run this demo, simply load the full script into an available script slot and tap "Test/Load".

```
// GPS Demo Script

// WARNING: This demo overwrites the contents of Script 50.
// WARNING: This demo does not contain any error checking and
// WARNING: may not work on all devices.

// Constants
var VK_RETURN = 0x0D; // Enter key

// Function to update position display when position
// event is signaled.
function UpdateGPSPosition()
{
    var p;
    var pl = Device.GPS.GetPosition();
    // Parse position literal object
    eval( "p="+pl );
    OS.Beep();
    // Update display message
    CETerm.Message.Text = "Lat and Long will display values when" +
                           " GPS obtains a location fix." +
                           " Initial fix may take several minutes." +
                           "\nPress ENTER when done.\nLat:" +
                           p.latitude + "\nLong:" + p.longitude;

    // Reschedule event listener
    var s = OS.Event.SetListener( "GPSPositionUpdateEvent",
                                  "UpdateGPSPosition();", 300000 );
}

// Function to cleanup GPS display when done
function CleanupGPS()
{
    var g = Device.GPS;
    var m = CETerm.Message;
    var e = OS.Event;
    var k = Device.Keyboard;

    // Close GPS device
    g.Close();

    // Hide display
    m.IsVisible = false;

    // Remove event listener
    e.ClearListener( "GPSPositionUpdateEvent" );

    // Delete event
    e.Delete( "GPSPositionUpdateEvent" );

    // Remove hot-key assignment
    k.DeleteHotKey( "IDA_SCRIPT_50" );
}


// Function which initializes demo
function GPSDemo()
{
  var s;    // status
  var g = Device.GPS;
  var m = CETerm.Message;
  var e = OS.Event;
  var k = Device.Keyboard;

  // Create event for GPS position update
  var ep = e.Create( "GPSPositionUpdateEvent", false );
```

```
    // Assign initial listener script for GPS position event
    s = e.SetListener( ep, "UpdateGPSPosition();", 300000 );

    // Assign event to GPS object
    g.NewLocationDataEvent = ep;
    g.MaximumAge = 5000;

    // Open GPS device
    s = g.Open();

    // Prepare message display
    m.AbortButtonVisible = false;
    m.Title = "GPS Demo";
    m.Text = "Initializing, please wait...";
    m.IsVisible = true;

    // Create global hot-key to run Script 50 for cleanup
    k.AssignHotKey( VK_RETURN, 0, "IDA_SCRIPT_50" );

    // Put cleanup script invocation into slot 50
    CETerm.SetProperty( "app.script.50", "CleanupGPS()" );
  }

  // Invoke GPSDemo function to start demo
  GPSDemo();
```

3.9 THE KEYBOARD OBJECT

The `Keyboard` object provides access to keyboard operations. The `Keyboard` object is a property of the `Device` object; `Device.Keyboard`. The `Keyboard` object can be used to simulate hardware keyboard actions to other applications or to CETerm if required. Simulated key events should not be used if CETerm Action codes can achieve the same result.

The `Keyboard` object can also create global “hot-keys” which can be used to activate CETerm actions or scripts. Global hot-keys are recognized regardless of which program is active and in the foreground. Global hot-keys can be an important feature when using CETerm as a lock-down shell and program launcher.

Methods

The following methods are available

Method	**Action**
AssignHotKey	Assign a global hot-key.
DeleteHotKey	Delete a global hot-key.
DeleteAllHotKeys	Delete all global hot-keys.
Enable	Enable or disable the keyboard.
IsEnabled	Check the keyboard enabled state.
IsHotKey	Check if a global hot-key has been assigned.
IsKeyDown	Check if a state key was down for the last key input.
IsKeyDownNow	Check if a key is currently down.
IsKeyToggled	Check if a state key is toggled on.
SimulateKeyDown	Simulate the press of a hardware key.
SimulateKeyUp	Simulate the release of a hardware key.
SimulateKeyPress	Simulate the press and release of a key and specify generated text.

status = AssignHotKey (vkCode, keyModifiers, idaCode)

Assign a CETerm action to a global hot-key. The vkCode is an integer "virtual-key code" between 1 and 254. See Appendix 5 for virtual-key values. The keyModifiers specify if the Alt, Ctrl, Shift, or Windows key must also be pressed with the activating key. See Appendix 4 for key modifier values. The idaCode is a CETerm IDA Action Code symbolic name as a text string. Each hot-key **must** use a unique IDA Action Code. Often, the IDA code will activate a CETerm script to perform multiple operations. If multiple hot key combinations must perform the same action, you can use separate but identical scripts. See Appendix 1 for IDA values. Return 0 for success, 1 if replaced an existing hot-key, or a negative value for failure. Use the `Keyboard` property LastError to get additional error information.

A global hot-key is recognized regardless of the foreground application, so it can be used to activate CETerm actions even when another program is visible.

WARNING: On Windows Mobile devices, the CETerm setting "Disable Windows Action Keys" will also disable all hot-keys in CETerm.

status = DeleteHotKey (idaCode)

Delete the global hot-key with the specified IDA action. Return 0 if deleted the hot-key, -1 if the key was not found, other negative value for failure. Use the `Keyboard` property LastError to get additional error information.

status = DeleteAllHotKeys ( )

Delete all global hot-keys. Return 0.

status = Enable (enabled)

Enable or disable the hardware keyboard. If enabled is true the keyboard is enabled, if false it is disabled. Not all devices support this action. Return the new keyboard state.

WARNING: If the keyboard is disabled, it may render the device unusable and require a system reset. Design your scripts defensively to prevent unwanted conditions.

status = IsEnabled ( )

Check the keyboard enabled state. Returns true if enabled, false if disabled. Not all devices support this action.

status = IsHotKey (idaCode)

Check if a global hot-key has been assigned with the specified IDA code. Return true if hot-key was assigned, false otherwise.

status = IsKeyDown (vkCode)

Check the status of the specified key. Return true if key was down, false if up. Note that this state is updated by Windows only when key events are processed. Use IsKeyDownNow() to check the instantaneous state of a key. See Appendix 5 for VK values.

status = IsKeyDownNow (vkCode)

Check the current status of the specified key. Return true if key is down, false if up. This instantaneous check is useful to detect when a user releases a key. See Appendix 5 for VK values.

status = IsKeyToggled (vkCode)

Check the toggle status of the specified key. Return true if key is toggled on, false if toggled off. This check is used for VK_CAPITAL (0x14) and VK_NUMLOCK (0x90) keys only.

status = SimulateKeyDown (vkCode)

Simulate the press of a hardware key. The vkCode is an integer “virtual-key code” between 1 and 254. See Appendix 5 for virtual-key values. This key event will be received by the current active (foreground) window which may not be CETerm. The key down action may also generate text input to the application. Typically SimulateKeyDown() is followed by a SimulateKeyUp(). Return 0 on success, -1 for invalid VK code.

status = SimulateKeyUp (vkCode)

Simulate the release of a hardware key. The vkCode is an integer “virtual-key code” between 1 and 254. See Appendix 5 for virtual-key values. This key event will be received by the current active (foreground) window which may not be CETerm. Typically SimulateKeyUp() follows a SimulateKeyDown() call. Return 0 on success, -1 for invalid VK code.

status = SimulateKeyPress (vkCode, keyStateFlags, text)

Simulate the press and release of a key and specify generated text. The vkCode is an integer “virtual-key code” between 1 and 254. See Appendix 5 for virtual-key values. The keyStateFlags indicate the simulated state of modifier keys. See Appendix 4 for key state flag values. The text is an arbitrary text string that will be sent to the application together with the key events. This key event will be received by the current active (foreground) window which may not be CETerm. SimulateKeyPress() should be used when you need to simulate text sent to an independent application. Return 0 on success, -1 for invalid VK code.

Properties

The `Keyboard` object has the following properties.

Property	Description	Values
LastError	Returns the last error value associated with the `Keyboard` object.	unsigned integer

Example

The following example shows how a global hot-key may be assigned to cycle between several running programs. This example also shows the use of several `OS.Window` methods. The hot-key invokes a CETerm script that will check which application is visible and switch to the next. The script is shown first. It must be imported into any CETerm script slot and marked "Load at Startup".

```
// Switch between multiple applications
// Each appX name can be a regular expression or string
// Can be any number of applications named in arguments
function ToggleApplications( app1, app2 )
{
    // Find current application
    var i;
    var top = OS.Window.GetTopmost();
    var toptext = OS.Window.GetText( top );
    var topindex = -1;
    // DEBUG: OS.Alert("toptext='" + toptext + "'");
    // Find argument that matches current foreground application
    // If no match, will switch to app1.
    for (i=0; i < arguments.length; ++i)
    {
        if (toptext.match( arguments[i] ))
        {
            topindex = i;
            break;
        }
    }

    // Find next application index
    var nextindex = (topindex + 1) % arguments.length;

    // DEBUG: OS.Alert("topindex=" + topindex +
    //                 " nextindex=" + nextindex +
    //                 " arg=" + arguments[nextindex] );

    // Find next window
    var wa = eval( OS.Window.GetList() );
    for (i=0; i < wa.length; ++i)
    {
        // DEBUG: OS.Alert( "text='" + wa[i].text + "'" );
        if (wa[i].text.match( arguments[nextindex] ))
        {
            OS.Window.SetTopmost( wa[i].hwnd );
        }
    }
}
```

The next step is to define the script that is run when the hot-key is pressed. This script invokes `ToggleApplications.` Let's assume this script is in Script #8.

```
// Switch between CETerm, Calculator, and Media Player.
ToggleApplications( /ceterm/i, /calc/i, "Media Player" );
```

The last step is to assign the hot-key to run Script #8. This can be placed in any script slot. It may be marked “Load at Startup” or the key could be assigned as part of other initialization scripts. For this example, Shift+Right_Arrow is the hot-key combination.

```
// Assign hot-key
var vk_rightarrow = 0x27;
var modifier_shift = 0x4;
var k = Device.Keyboard;
var result = k.AssignHotKey( vk_rightarrow, modifier_shift,
                             "IDA_SCRIPT_8" );
if (result != 0)
{
    OS.Alert( "AssignHotKey failed e=" + k.LastError );
}
```

3.10 THE MESSAGE OBJECT

The `Message` object provides feedback to the user while a script is running. The `Message` object is a property of the `CETerm` object; `CETerm.Message`. This object displays a dialog with a text message, an optional progress bar, and an optional script cancellation button. The progress value can be set by the script as tasks are completed, or it can run at a constant rate to show activity to the user.

Methods

The `Message` object has no methods.

Properties

The `Message` object is controlled through read-write properties. Setting a property will change the message dialog appearance.

Property	Description	Values

Property	Description	Values
AbortButtonVisible	If true, a script abort button is visible. Taping this button will abort the current script execution.	true, false
IsVisible	If true, message dialog is visible.	true, false
Progress	Current progress value in percent.	0 to 100
ProgressRate	Rate of change of progress bar. In units of percent per second.	1 to 100
ProgressRunning	If true, Progress increases at ProgressRate	true, false
ProgressVisible	If true, progress bar is shown	true, false
Text	Text of message.	text
Title	Text in message box title bar.	text
Timeout	Visibility timeout for dialog. After this interval, message dialog is closed. A value of 0 disables this timeout.	0 to 9999

3.11 THE NETWORK OBJECT

The `Network` object provides access to the network features such as hostname resolution, ping, and FTP. The `Network` object is a property of the `OS` object; `OS.Network`.

Methods

The following methods are available

Method	Action
DNSLookup	Lookup the IP address of a given hostname.
Ping	Test whether a host is reachable across an IP network.

ipAddress = DNSLookup (hostname)

Return the IP address corresponding to the hostname as a string. Returns null if cannot resolve the IP address. Use the `Network` property LastDNSError to get additional error information.

status = Ping (hostname, timeout)

Sends an ICMP packet to the hostname and waits timeout milliseconds for a reply. The hostname may be a text name or an IP address. Return the round-trip response time in milliseconds. Return 0 for no response, or a negative value on error. Use the `Network` property LastPingError to get additional error information.

The Ping command is synchronous, so no other CETerm operations occur while it is active. You should minimize the timeout value. The property MaximumPingTimeout limits the timeout you can specify in the Ping command.

Properties

The `Network` object has the following properties.

Property	**Description**	**Values**
FTP	Returns the FTP object. This object provides access to FTP operations. (read only)	object
LastDNSError	Returns the last error value associated with the DNSLookup method. (read only)	unsigned integer
LastPingError	Returns the last error value associated with the Ping method. (read only)	unsigned integer
LastPingHostName	Returns the hostname used in the last Ping operation. (read only)	string
LastPingIPAddress	Returns the IP address used in the last Ping operation. (read only)	string
LastWSAError	Returns the last error value associated with any Windows socket operations. (read only)	unsigned integer
MaximumPingTimeout	Controls the maximum timeout for the Ping command. The default value is 3000 milliseconds. Use caution when increasing this value due to the delays it may produce.	integer

3.12 THE PROCESS OBJECT

The `Process` object provides access to running Windows processes. Processes can be started, killed, and listed. The `Process` object is a property of the `OS` object; `OS.Process`.

Methods

The following methods are available

Method	Action
ExecuteAction	Run a program or open a file via the specified action.
GetList	Get a list of the running processes
Kill	Stop (Kill) a running process.
WaitForExit	Wait for a running process to exit.

status = ExecuteAction (file, arguments, verb)

Run a program using the specified action verb. The file specifies either an executable program that will be run or a general document file that will be processed according to the specified verb. The arguments are specified as a text string and are passed to the program that is run. Arguments are separated by spaces. Use quotes if an argument contains spaces. If you want to include double quotation marks as part of an argument, these must be enclosed by two sets of double quotation marks. For example the argument string

```
var args = 'first "arg with spaces" """quotes part of arg""" last';
```

specifies 4 arguments: `first`, `arg with spaces`, `"quotes part of arg"`, and `last`.

Normally, the verb is “open” for executable files, but it may be “edit” or “print” to invoke those actions on a document file. Return 0 on success or non-zero otherwise. Use the `Process` property LastError to get additional error information. Use the property LastExecuteProcess to get the process ID of the last process started.

processList = GetList ( )

Return a list of currently running processes. The returned list is in the form of a JavaScript array literal […] which contains JavaScript object literals {…}

containing information about each process. See Section 2.8 for details about handling JavaScript literals. Return null if error. Use the `Process` property LastError to get additional error information.

The returned literal has the following format:

```
[ {processID:0x17fdf002, threads:2, name:"NK.EXE"},
{processID:0x17fcb266, threads:6, name:"filesys.exe"},
{processID:0xb7f67176, threads:85, name:"device.exe"},
{processID:0x97a5a6e6, threads:10, name:"gwes.exe"},
{processID:0xf79f79aa, threads:4, name:"explorer.exe"},
{processID:0xb79f7e32, threads:7, name:"services.exe"},
{processID:0x77452dda, threads:2, name:"CETerm.EXE"} ]
```

status = Kill (processID)

Terminates a currently running process identified by the process ID. Return 0 on success or non-zero otherwise. Use the `Process` property LastError to get additional error information.

exitvalue = WaitForExit (processID, timeout)

Wait timeout milliseconds for a currently running process to terminate. Return the exit value of the process. Use the `Process` property LastError to determine if the call timed out, there was an error, or the process terminated normally.

While waiting for the process, CETerm is prevented from performing any other actions. Keep the timeout to less than a couple of seconds. It is better to use the Event.SetProcessListener() method to detect the termination of a process.

Properties

The `Process` object has the following properties.

Property	**Description**	**Values**
LastError	Returns the last error value associated with any Process operation. (read only)	unsigned integer
LastExecuteProcess	Returns the process ID of the last process created by ExecuteAction. (read only)	unsigned integer

3.13 THE REGISTRY OBJECT

The `Registry` object provides access to the Windows registry. The registry is a form of database on Windows devices which holds the device configuration. The registry has a hierarchical structure. The "keys" are similar to file folders and the "values" inside a key are similar to files in a folder.
For a better understanding of the Windows registry, you can search for information at msdn.microsoft.com with the keywords "using registry ce".

Several methods require a "fully qualified" value name which contains the full key hierarchy, begins with a "root" key, and ends with the value name. This fully qualified value name is similar to a file name with the full path. The `Registry` object is a property of the `OS` object; `OS.Registry`.

WARNING: Altering the registry can make your device unusable. Be sure you understand the effect of changing values.

Methods

The following methods are available

Method	Action
DeleteKey	Delete an existing key.
DeleteValue	Delete an existing value.
EnumerateKeys	Get all sub-key names of a specified key.
EnumerateValues	Get all value names of a specified key.
FlushKey	Issue the RegFlushKey command.
GetValueType	Get the data type of a value.
ReadValue	Read a value from a key.
ReadValueVBArray	Read a value from a key and return as a Visual Basic array.
WriteValue	Write a value to a key.

status = DeleteKey (keyname)

Deletes an existing key and all values. Returns 0 for success or non-zero for an error. Delete will fail if a key has sub-keys.

status = DeleteValue (keyname, valuename)

Deletes the specified value in an existing key. Returns 0 for success or non-zero for an error.

keylist = EnumerateKeys (keyname)

Return a list of sub-keys of the specified key. See Appendix 4 for key names and definitions. Each sub-key in the list is separated by the current StringSeparator property value.

keylist = EnumerateValues (keyname)

Return a list of values of the specified key. See Appendix 4 for key names and definitions. Each value name in the list is separated by the current StringSeparator property value.

status = FlushKey (keyname)

Performs a Windows CE “RegFlushKey” on the specified key. Some older devices use this to trigger a save of the current registry to persistent memory. Do not use FlushKey unless directed by the device manufacturer. Returns 0 for success, non-zero for error.

type = GetValueType (valuename)

Gets the data type for the specified value. Use a fully qualified value name that starts with a root key. Returns 0 for success or non-zero for an error.

data = ReadValue (valuename)

Read the data from the specified value. Use a fully qualified value name that starts with a root key. Binary values are returned as a list of comma separated hexadecimal digits. MULTI_SZ strings are separated with the current StringSeparator property value.

data = ReadValueVBArray (valuename)

Read the data from the specified value. Return the data as a Visual Basic array. Use a fully qualified value name that starts with a root key. It is usually best to use the ReadValue method and split the values using JavaScript. In rare circumstances a true array may be needed. It is not possible to return a

JavaScript array, but it is easy to convert a Visual Basic array into a JavaScript array using the VBArray object. For example:

```
var valuename = "HKLM\\Comm\\PY21BG1\\Parms\\TcpIp\\DhcpDNS";
var vbarray = new VBArray( OS.Registry.ReadValueVBArray( valuename );
var jsarray = vbarray.toarray();
```

status = WriteValue (valuename, valuedata, datatype)

Write the specified value. Use a fully qualified value name that starts with a root key. WriteValue will create the containing key if it does not exist. Binary values are submitted as a list of comma separated hexadecimal digits. MULTI_SZ strings are separated with the current StringSeparator property value. Returns 0 for success, non-zero for error. See Appendix 4 for root key names and datatype definitions. Common datatypes are “REG_SZ” for a string and “REG_DWORD” for a DWORD value.

Properties

The `Registry` object has the following property.

Property	Description	Values
StringSeparator	Text string that separates MULTI_SZ values. Default: "<|>"	text

3.14 THE SCREEN OBJECT

The `Screen` object gives access to a session terminal emulation screen. The `Screen` object is a property of the `Session` object; `CETerm.Session(i).Screen`. This section documents the methods and properties of the `Screen` object.

To access text in a browser page, use the `Browser.Document` reference and read the text directly from the desired page element.

Methods

The following methods are available

Method	Action
GetText	Get all text from start location to end location
GetTextLine	Get all text on a line
GetTextRect	Get a rectangle of text

text = GetText (startRow, startColumn, endRow, endColumn)

Return the requested range of text. Each line will be separated by the TextLineSeparator property value. If the session is not connected the JavaScript null value is returned. End coordinates of -1 will use the maximum valid value.

text = GetTextLine (row)

Return the requested row of text. If the session is not connected the JavaScript null value is returned. The row range is from 1 to the maximum row number.

text = GetTextRect (startRow, startColumn, endRow, endColumn)

Return the requested rectangle of text. Each line fragment will be separated by the TextLineSeparator property value. If the session is not connected the JavaScript null value is returned. End coordinates of -1 will use the maximum valid value.

Properties

The `Screen` object has the following properties.

Property	Description	Values
Rows	Number of rows in screen. (read only)	1-50
Columns	Number of columns in screen. (read only)	1-132
CursorRow	Current row containing cursor. Setting this value will change the cursor location.	1 to Rows
CursorColumn	Current column containing cursor. Setting this value will change the cursor location.	1 to Columns
DisplayStatus	IBM display status. (read only)	integer, see Appendix 4
KeyboardState	Keyboard state. This applies to VT and	integer,

Property	Description	Values
	IBM emulation only. VT state can only be locked or unlocked. (read only)	see Appendix 4
TextLineSeparator	Text which separates every line in GetText methods.	Default: nothing

3.15 THE SERIALPORT OBJECT

The `SerialPort` object gives access to serial port functionality. The `SerialPort` objects are obtained from a `Device` object method; `Device.SerialPort(i)` where `i` is `0` through `9`. This section documents the methods and properties of the `SerialPort` object. The `SerialPort` object has been used to integrate devices such as tethered scanners, Bluetooth scanners, and RFID readers into CETerm. See Chapter 5 for additional details about using the `SerialPort` object.

Methods

The following methods are available

Method	Action
CancelWaitForEvent	Stop listening for a serial port event.
ClearBreak	Clear the break condition.
ClearError	Clear any error conditions and return status information.
Close	Close the serial port and terminate communications.
Open	Open the serial port for communication.
PurgeQueues	Discard content of input and output queues.
Read	Read up to a maximum number of bytes.
ReadByte	Read one byte
ReadTillByte	Read up to a maximum number of bytes, or a specified byte value.
SetBreak	Set the break condition.
SetQueueSizes	Set the size of input and output queues.
WaitForEvent	Start listening for a serial port event.
Write	Write a string of characters.
WriteByte	Write a single byte value
WriteNULL	Write a number of NULL (0) bytes
WriteUrgent	Write a single urgent byte at the front of the output queue.

status = CancelWaitForEvent ( )

Cancel the active event listener for the port. If canceled, changes in port status will not be reported through the event handler OnSerialPortEvent. Return 0 on success, non-zero for failure.

status = ClearBreak ( )

Clear the break condition on the port. Return 0 on success, non-zero for failure. Use the `SerialPort` property LastError to get additional error information.

status = ClearError ( )

Clear the error condition and re-enable I/O operations. Return information about the error condition and current port status as a JavaScript literal object. Return null if the port is not open or other failure. More details can be found by searching msdn.microsoft.com with the keyword “clearcommerror”.

The general format of the status is:

```
{"errorType":1, "CTSHold":false, "DSRHold":false,
"RLSDHold":false, "XOFFHold":false, "XOFFSent":false,
"EOF":false, "TXIM":false, "inputQueue":0, "outputQueue":0}
```

status = Close ( )

Close the serial port. Any active event listener is canceled. Return 0 on success, non-zero for failure. Use the `SerialPort` property LastError to get additional error information.

status = Open (access)

Open the serial port for communications. The access parameter code specifies read and/or write access. Return 0 on success, non-zero for failure. Use the `SerialPort` property LastError to get additional error information. See Appendix 4 for access definitions.

status = PurgeQueues (mode)

Discard content from input and output queues. Return 0 on success, non-zero for failure. Use the `SerialPort` property LastError to get additional error information. See Appendix 4 for mode definitions.

data = Read (maxCount)

Read up to maxCount bytes from the port. Return data as a string. Return null if no data or error. This method blocks until the maximum bytes are read or the read timeout expires. Data are converted from bytes to a wide character string by `mbstowcs` using the current locale. Use the `SerialPort` property LastError to get additional error information.

data = ReadByte ( )

Read a single byte from the port. Return data as an integer value between 0 and 255. Return negative number if failure; -1 – failure, -2 – timeout. This method blocks until the byte is read or the read timeout expires. Use the `SerialPort` property LastError to get additional error information.

data = ReadTillByte (maxCount, byteValue)

Read up to maxCount bytes from the port. Return data as a string. Return null if no data or error. This method blocks until the maximum bytes are read, or a byte with the specified integer value is read, or the read timeout expires. The suggested technique to use ReadTillByte is to call it after the EV_RXFLAG event is signaled through OnSerialPortEvent with desired character as the event character. Use the `SerialPort` property LastError to get additional error information.

status = SetBreak ( )

Set the break condition on the port. Suspends character transmissions and enters the break state. Return 0 on success, non-zero for failure. Use the `SerialPort` property LastError to get additional error information.

status = SetQueueSizes (inputQueueSize, outputQueueSize)

Set the size of input and output queues. Return 0 on success, non-zero for failure. Use the `SerialPort` property LastError to get additional error information. More details can be found by searching msdn.microsoft.com with the keyword “setupcomm”.

status = WaitForEvent ( )

Enable an event listener for the port. The listener waits for events that are specified in the property EventMask. If the event is signaled by the port, the event handler OnSerialPortEvent is invoked. The event handler is invoked only once for each call of WaitForEvent, but the handler parameters may indicate multiple event conditions. Return 0 on success, non-zero for failure.

status = Write (data)

Write the data string to the port. Data are converted from wide characters to bytes with `wcstombs` using the current locale. Return number of bytes written for success, negative value for failure. Use the `SerialPort` property LastError to get additional error information.

status = WriteByte (byteValue)

Write a single byte with integer value byteValue to the port. Return 1 for success, negative value for failure. Use the `SerialPort` property LastError to get additional error information.

status = WriteNULL (nullCount)

Write NULL (0) bytes to the port. Write nullCount NULL bytes to the port. Return number of bytes written for success, negative value for failure. Use the `SerialPort` property LastError to get additional error information. Where needed, this method may be used to “wakeup” an attached device prior to sending commands.

status = WriteUrgent (byteValue)

Write a single byte with integer value byteValue to the port. This places the byte ahead of any pending data in the output buffer. Return 1 for success, negative value for failure. Use the `SerialPort` property LastError to get additional error information. More details can be found by searching msdn.microsoft.com with the keyword “transmitcommchar”.

Properties

Many of the `SerialPort` properties correspond directly to Windows serial port configuration values. More details can be found by searching msdn.microsoft.com with the keywords “commtimeouts structure ce” and “DCB structure ce”.

In general, you should set all necessary properties before opening the serial port. However, you may alter the properties of an open port by assigning new values. See Chapter 5 for additional details about using the `SerialPort` object.

The `SerialPort` object properties are listed in the following tables. The first table contains general purpose properties, the second contains device control (DCB) properties and the third contains timeout properties. Values marked (D) indicate the default setting.

General Property	**Description**	**Values**
EventMask	Events monitored by WaitForEvent. Default is none.	integer flags, see Appendix 4
IsOpen	Returns true if port is open. (read only)	true, false
LastError	Returns the last error value associated with any SerialPort operation. (read only)	unsigned integer
ModemStatus	Modem status. (read only)	integer flags, see Appendix 4
PortIndex	Index of this SerialPort object. (read only)	integer
PortName	Name of the serial port device. Default value is “COMx:” where x is the PortIndex.	text

DCB Property	**Description**	**Values**
BaudRate	Baud rate of device. Default 115200.	integer, see Appendix 4
CheckParity	Perform parity checking.	true, false(D)
CTSOutputFlowControl	Monitor CTS signal for output flow control.	true, false(D)
DataBits	Bits per byte.	integer (D:8)
DiscardReceivedNULL	Discard any null bytes received.	true, false(D)
DSRInputControl	Monitor DSR signal for input flow.	true, false(D)

DCB Property	**Description**	**Values**
DSROutputFlowControl	Monitor DSR signal for output flow control.	true, false(D)
DTRControlMode	DTR signal mode. Default is “enable when open” (0x1).	integer, see Appendix 4
EventCharacter	Character that triggers EV_RXFLAG event.	integer (D:0x0)
OutputContinueOnXOFF	Continue output when XOFF has been sent to restrict input.	true(D), false
ParityMode	Parity scheme to use when parity checking is enabled. Default is “none” (0x0).	integer, see Appendix 4
RTSControlMode	RTS signal mode. Default is “enable when open” (0x1).	integer, see Appendix 4
StopBits	Number of stop bits to use. Default value is “one stop bit” (D:0x0)	integer, see Appendix 4
XONCharacter	Value of XON character. (D:0x11)	integer
XONLowerLimit	Consumed space threshold in input buffer below which flow control is relaxed to allow additional input. Input flow control may be XON/XOFF, RTS, or DTR.	integer (D:100)
XOFFCharacter	Value of XOFF character. (D:0x13)	integer
XOFFInputFlowControl	Enable XON/XOFF control for reception. Send XOFFCharacter when XOFFUpperCushion is reached. Send XONCharacter when XONLowerLimit is reached.	true(D), false
XOFFOutputFlowControl	Enable XON/XOFF control for transmission. Stop transmission when XOFFCharacter is received, re-start when XONCharacter is received.	true(D), false
XOFFUpperCushion	Minimum available space in input buffer allowed before flow control is activated to stop additional input. Input flow control may be XON/XOFF, RTS, or DTR. (D:0x0)	integer (D:100)

Timeout Property	Description	Values
ReadIntervalTimeout	Maximum milliseconds allowed between the arrival of two bytes. If this time is exceeded the read call will return. A value of 0 means not used. With the special value 0xffffffff, the read operation will return immediately with all bytes already received if the ReadTotalTimeoutConstant and ReadTotalTimeoutMultiplier are both zero.	integer (D:0xffffffff)
ReadTotalTimeoutConstant	Constant milliseconds time added to compute total timeout.	integer (D:0)
ReadTotalTimeoutMultiplier	Milliseconds factor multiplied times the number of bytes requested in the read request. This is added to the ReadTotalTimeoutConstant to yield the total read timeout.	integer (D:0)
WriteTotalTimeoutConstant	Constant milliseconds time added to compute total timeout.	integer (D:0)
WriteTotalTimeoutMultiplier	Millisecond factor multiplied times the number of bytes to be written. This is added to the WriteTotalTimeoutConstant to yield the total write timeout.	integer (D:0)

See a complete discussion of timeout properties, special values, and special behaviors by searching msdn.microsoft.com for “commtimeouts structure ce”.

3.16 THE SESSION OBJECT

The `Session` object gives access to session state. The `Session` object is obtained from a `CETerm` object method; `CETerm.Session(i)`. This section documents the methods and properties of the `Session` object.

Methods

The `Session` object has no methods.

Properties

The `Session` object has several read only properties.

Property	Description	Values
Browser	Returns browser object. (read only)	object
IsConnected	Returns true if session is connected. (read only)	true, false
Screen	Returns screen object. (read only)	object

3.17 THE TEXTINPUT OBJECT

The `TextInput` object provides user input in a script. This object displays a dialog with a text message, an input field, a Cancel button and an OK button. The `TextInput` object is a property of the `CETerm` object; `CETerm.TextInput`.

Methods

The following methods are available

Method	Action
GetInput	Get input from the user

result = GetInput ( )

GetInput displays the user input dialog. Returns 1 for successful input, 0 if input is canceled by the user or -1 if there was an error. A default response may be set in the Input property prior to calling GetInput. If no default is desired, be sure to clear Input prior to calling GetInput.

Properties

The `TextInput` has the following properties.

Property	Description	Values
Input	Can be pre-set with default response before calling GetInput. If GetInput returns 1, contains the user input.	text
PasswordMode	If true, input is shown as * characters.	true, false
Prompt	Text prompt message for user.	text
Title	Text in message box title bar.	text

3.18 THE WINDOW OBJECT

The `Window` object provides access to the displayed windows of running processes. This object can be used to find applications and send messages to those applications. This feature allows CETerm to control and cooperate with other applications. The `Window` object is a property of the `OS` object; `OS.Window`.

WARNING: Altering window visibility and input states can make your device unusable and require a device reset. Be sure you understand the effect of changing values.

Methods

The following methods are available

Method	Action
EnableInput	Enable or disable input to a window.
Find	Find a named window if it exists.
GetDesktop	Get the handle for the desktop window.
GetList	Get a list of top-level windows.
GetParent	Get the parent of the specified window.
GetRelative	Get a relative (child or sibling) of the specified window.
GetSelf	Get the handle of the top-level CETerm window.
GetText	Get the text of the specified window.

GetTopmost	Get the window with which the user is working.
IsEnabled	Check if window is enabled for input.
IsVisible	Check if window is visible.
IsWindow	Check if window handle is valid.
PostMessage	Post a message to a window.
SendMessage	Send a message to a window.
SetTopmost	Set a window as the current working window.
Show	Show or hide a window.

status = EnableInput (windowHandle, enabled)

Enable or disable input to a window. If enabled is true then mouse and keyboard input is enabled. Return true if window was previously disabled or false if window was previously enabled. Check the `Window` property LastError to determine the success of the method.

windowHandle = Find (windowClass, windowName)

Find the handle of the top-level window that matches the given window class or window name. Either argument may be an empty string. Return the non-zero window handle or 0 if not found. Use the `Window` property LastError to get additional error information.

windowHandle = GetDesktop ( )

Get the handle of the desktop window.

windowList = GetList ( )

Return a list of current top-level windows. The returned list is in the form of a JavaScript array literal […] which contains JavaScript object literals {…} containing information about each window. See Section 2.8 for details about handling JavaScript literals. Return null if error. Use the `Window` property LastError to get additional error information.

The returned literal has the following format:

```
[ {hwnd:0x7c010680, processID:0x0, text:"CursorWindow"},
{hwnd:0x7c012d70, processID:0xf79f79aa, text:""},
{hwnd:0x7c01cf20, processID:0x77452dda, text:"Edit Script 6"},
{hwnd:0x7c01bb70, processID:0x77452dda, text:"Scripting"},
{hwnd:0x7c0189c0, processID:0x77452dda, text:"S1 - Configure"},
{hwnd:0x7c015c50, processID:0x77452dda, text:"CETerm - S1"},
{hwnd:0x7c012320, processID:0xb7f67176, text:"Input Panel"},
{hwnd:0x7c011d40, processID:0xb79f7e32, text:"WinCENotify"} ]
```

windowHandle = GetParent (windowHandle)

Get the handle of the parent window. If the window is a child window, the return value is a handle to the parent window. If the window is a top-level window, the return value is a handle to the owner window. If the window is a top-level unowned window or if the method fails, the return value is 0. Use the `Window` property LastError to get additional error information and differentiate between failure and a top-level unowned window.

windowHandle = GetRelative (windowHandle, relation)

Get the handle of a window with the relation to the specified window. The relation values are integers and are listed in Appendix 4. Return the window handle or zero if no-such-window or method fails. Use the `Window` property LastError to get additional error information and differentiate between failure and no-such-window.

windowHandle = GetSelf ( )

Get the handle of the top-level CETerm window.

text = GetText (windowHandle)

Get the text of the specified window. Return the text of the window’s title bar or the text contents if the window is a control. Return null if function fails. Use the `Window` property LastError to get additional error information.

windowHandle = GetTopmost ( )

Get the window with which the user is working. This is also called the foreground window. May return 0 if no window is currently active. Use the `Window` property LastError to get additional error information.

status = IsEnabled (windowHandle)

Check the input status of the specified window. Return true if the window is accepting input or false if the window is not accepting input. Use the `Window` property LastError to get additional error information.

status = IsVisible (windowHandle)

Check the visibility state of the specified window. Return true if the window may be visible or false if the window is hidden. A window in the "visible" state may still be hidden from view by other windows. Use the `Window` property LastError to get additional error information.

status = IsWindow (windowHandle)

Check if the specified window handle is valid. Return true if the window handle identifies an existing window or false if the window does not exist. Use the `Window` property LastError to get additional error information.

status = PostMessage (windowHandle, message, wParam, lParam)

Post a message to the specified window. Return the status of the posting, not the status of processing the message. This feature is primarily intended to work with applications that document public messages that may be used for control or communication. Valid message values and parameters are not documented in this manual. Use the `Window` property LastError to get additional error information.

WARNING: Posting an ill-formed message may cause the receiving application to fail. There are very few messages that are valid from this method.

status = SendMessage (windowHandle, message, wParam, lParam)

Send a message to the specified window. Return the integer status of the message execution. This feature is primarily intended to work with applications that document public messages that may be used for control or communication. Valid message values and parameters are not documented in this manual. Use the `Window` property LastError to get additional error information.

WARNING: Sending an ill-formed message may cause the receiving application to fail. There are very few messages that are valid from this method. Beware that some messages return complex results and these are not returned to the caller and may cause CETerm to fail.

status = SetTopmost (windowHandle)

Set the specified window to be the topmost and active window. This is also called the foreground window. Return true on success or false on failure.

status = Show (windowHandle, visible)

Set the visibility state of the specified window. If visible is true, the window is set to the visible state. If visible is false, the window is hidden. Return true if the window was previously visible or false if the window was previously hidden.

Properties

The `Window` object has the following properties.

Property	**Description**	**Values**
LastError	Returns the last error value associated with any `Window` operation. (read only)	unsigned integer
SendMessageTimeout	Timeout in milliseconds of SendMessage method. Default value is 5000.	integer

4.0 CETerm Script Events

This section describes the script events within the CETerm script engine. These events provide ways to trigger event handlers when various conditions occur in CETerm. The event handlers are arbitrary scripts.

The event model in CETerm uses specific event handler names to bind events to handlers. If the event handler function (e.g., “OnBarcodeRead”) is defined in the script engine, it will be executed when the event occurs. There is no special command required to register or bind the function to the event. Event handlers can be re-defined at any time. If the handler is no longer needed, the function can be re-defined as empty.

Events play a very important role for scripting in CETerm. Just as in the standard web browser, a script cannot run continuously, or it will prevent user interaction and other program actions. The script engine acts like a “virtual user”. When a script is executing, CETerm will seem unresponsive. Typically, a script will do a little bit of work and then exit. This way, CETerm is always ready to respond to the user or host actions. Events and timers are used to start or re-start a script to do the next bit of work.

The “expect” script described in Section 2.6 is a good example of using a timer to automate multiple steps. The events described in this section are the second major technique for running a script to satisfy a condition.

Event	**Fired when…**
OnBarcodeRead	Barcode read.
OnDocumentDone	New web page loads.
OnIBMCommand	Receives special command in IBM data stream.
OnKeyboardStateChange	Keyboard state changes in TE session.
OnNavigateError	Web navigation fails.
OnNavigateRequest	Web navigation begins.
OnNetCheckFailed	Fails to complete network check to host.
OnProgramExit	CETerm exits.
OnProgramStart	CETerm first starts.
OnSerialPortEvent	Serial port status changes.
OnSessionConnect	Session connects to host.
OnSessionDisconnect	User disconnects session from host.
OnSessionDisconnected	Session disconnected by host.
OnSessionReceive	TE session receives data from host.
OnSessionSwitch	Active session changes.
OnStylusDown	Stylus tap on screen.
OnTriggerEvent	Hardware trigger status changes.
OnVTCommand	Receives special command in VT data stream.
OnWakeup	Device resumes after a suspend.

4.1 THE ONBARCODEREAD EVENT

The `OnBarcodeRead` event is fired when a barcode is successfully read. The handler can pre-process the data or check other conditions prior to passing it on to a TE or browser session.

Syntax

```
function OnBarcodeRead ( session, data, source, type, date, time
)
```

session – index of currently active session
data – barcode data
source – source of barcode. Typically a constant scanner name.
type – labeltype of barcode. See Appendix 3 for values.
date – date of barcode read.
time – time of barcode read.

Example

Several samples for OnBarcodeRead were given in Section 2.5. Following is an example that checks the RF connection before submitting the data to the host. This notifies the user that the barcode was not received by the host and instructs the user to return to RF coverage.

```
/* OnBarcodeRead */
function OnBarcodeRead ( session, data, source, type, date, time )
{
    // Check RF status
    var status = CETerm.GetProperty ( "device.rf.status" );
    if (status <= 0)
    {
        OS.Alert( "No RF signal detected.\n" +
                  "Barcode discarded.\n" +
                  "Return to RF coverage." );
        // Discard barcode
        return 1;
    }

    // Send barcode to emulator
    CETerm.SendText ( data, session );
```

```
        // Return 1 if handled data here
        return 1;
    }
```

4.2 THE ONDOCUMENTDONE EVENT

The `OnDocumentDone` event is fired when a new webpage has completed loading into a web browser session. The handler can add META tag definitions, examine or alter the Document Object Model (DOM), or add JavaScript methods to the page. This event allows CETerm to enhance a web page for mobile data collection that was not originally designed for such.

Syntax

```
function OnDocumentDone ( session )
```

session – index of browser session which completed page load.

Example

This example shows how several META tags can be added to a web page. We will add a “PowerOn” handler, a key remapping, and information item tags to position the RF indicator at a special location. The “PowerOn” handler is often used to navigate to a specific page, such as a login page, when the device resumes. The RF indicator tags will restore a specific location, but could be used to alter the RF indicator location depending on the current page.

```
/* OnDocumentDone */
function OnDocumentDone ( session )
{
  var b = CETerm.Session( session ).Browser;

  // Do not process the initial about:blank page
  if (!b.Document.URL.match("about:blank"))
  {
    // Add PowerOn META handler
    b.AddMetaItem( "PowerOn",
                   "Javascript:alert(\"My PowerOn\");" );

    // Insert new JavaScript function
    b.RunScript( "function myf1() {alert(\"F1 Function\");}" );

    // Add Key mapping to inserted function
    b.AddMetaItem( "OnKey_F1", "Javascript:myf1();" );

    // Position RF signal indicator
    b.AddMetaItem( "Signal", "x=195" );
```

```
        b.AddMetaItem( "Signal", "y=100" );

        // Update information items for location to take effect
        CETerm.PostIDA( "IDA_INFO_REFRESH", session );
      }
    }
```

4.3 THE ONIBMCOMMAND EVENT

The `OnIBMCommand` event is fired when a special extended command format is received in an IBM emulation session screen. Extended commands are documented elsewhere, but basically this event requires the two characters “#X” starting in the second column of the first row. The CETerm configuration option “Extended Commands” must be enabled for this event to fire. The handler script may perform any desired actions. Typically, the screen text contains additional information used by the handler.

Syntax

```
function OnIBMCommand ( session, command )
```

session – index of session receiving the command
command – specified command (e.g., “#X”)

Example

This example looks at the data following the command and activates an FTP file transfer.

```
    /* OnIBMCommand */
    function OnIBMCommand ( session, command )
    {
        // Get full line from screen
        // Expect: #X|FTP|myserver|localfilename|remotefilename
        var line1 = CETerm.Session( session ).Screen.GetTextLine( 1 );
        var args = line1.split( "|" );

        // Activate FTP
        if ("FTP" === args[1])
        {
            var ftp = OS.Network.FTP;
            if (0 === ftp.Login( args[2], "ftpuser", "secret" ))
            {
                ftp.PutFile( args[3], args[4] );
                ftp.Logout();
            }
        }
        // Submit screen to move to next action
        CETerm.PostIDA( "IDA_ENTER", session );
    }
```

4.4 THE ONKEYBOARDSTATECHANGE EVENT

The `OnKeyboardStateChange` event is fired when the state of the keyboard is changed by a user or host action. Typically this event is only used with IBM sessions. A VT session may generate this event only if custom escape sequences lock the keyboard.

Syntax

```
function OnKeyboardStateChange ( session, state )
```

session – index of currently active session
state – new keyboard state

Example

This example disables the scanner when the keyboard enters the locked mode.

```
    /* OnKeyboardStateChange */

    var IBM_KEYBOARD_HARDWARE_ERROR = 0;
    var IBM_KEYBOARD_NORMAL_LOCKED = 1;
    var IBM_KEYBOARD_NORMAL_UNLOCKED = 2;
    var IBM_KEYBOARD_POWER_ON = 3;
    var IBM_KEYBOARD_PRE_HELP_ERROR = 4;
    var IBM_KEYBOARD_POST_HELP_ERROR = 5;
    var IBM_KEYBOARD_SS_MESSAGE = 6;
    var IBM_KEYBOARD_SYSTEM_REQUEST = 7;

    var PreviousKeyboardState = [0,0,0,0,0,0];

    function OnKeyboardStateChange( session, state )
    {
        // Disable scanner if keyboard is locked.
        if (state === IBM_KEYBOARD_NORMAL_UNLOCKED)
        {
            if (PreviousKeyboardState[session] !==
                          IBM_KEYBOARD_NORMAL_UNLOCKED)
            {
                CETerm.PostIDA( "IDA_SCAN_RESUME", 0 );
            }
        }
        else
        {
            if (PreviousKeyboardState[session] ===
                          IBM_KEYBOARD_NORMAL_UNLOCKED)
            {
                CETerm.PostIDA( "IDA_SCAN_SUSPEND", 0 );
            }
        }

        // Save new state
        PreviousKeyboardState[session] = state;
    }
```

4.5 THE ONNAVIGATEERROR EVENT

The `OnNavigateError` event is fired if the browser fails to complete a navigation. Typically, the error handler will redirect the web browser to a "file:" URL on the device for error recovery. This event may fire if the device loses RF coverage during a navigation or the web server crashes. It is a good practice to use the CETerm "Check Network Before Send" feature to validate RF coverage prior to submitting the navigation request and use the `OnNavigateError` for additional error handling.

Syntax

```
function OnNavigateError ( session, params )
```

session – index of browser session which failed to navigate.
params – navigation error parameters, including the error number and URL.

The params argument is formatted as URL parameters and has the form:

```
error=0x800C0005&url=http://192.168.1.20/application.exe?state=3&scan=0
```

Everything after `url=` in the params argument is the URL that failed to navigate, along with all the parameters of that URL. The `error` values are standard Microsoft browser status codes and are defined in Appendix 4.

Example for Windows CE 5.0 devices

This example shows how to redirect a web browser to a static URL on the device.

```
    /* OnNavigateError */
    function OnNavigateError ( session, params )
    {
        // Save params in text 3x where x is session index
        // This is required by CE 5.0 devices which do not pass
        // parameters to a "file:" URL.
        CETerm.SetProperty( "app.usertext.3" + session, params );

        // Navigate to static error page
        var b = CETerm.Session( session ).Browser;
        b.Navigate( "file:///error.htm" );
    }
```

Note the `CETerm.SetProperty()` call. This method saves the params in persistent memory for later use by the "error.htm" web page. The reason to do this is because the parameters are discarded by the Windows CE browser when navigating to a "file:" resource. The error web page can retrieve the params using:

```
    var property = "app.usertext.3" + external.sessionindex;
    var params = external.CETerm.GetProperty ( property );
```

Using the URL, the error page can re-attempt the navigation or decide on other error recovery. Please note that the "User Text x" is used for several purposes in CETerm, including key remapping. Be sure that this use does not collide with other uses in your configuration.

Example for Windows Mobile devices

Handheld devices using Windows Mobile can use a different technique to pass on the params URL. For these devices, the parameters of a “file:” URL are available within the browser. The error parameters can simply be passed on to the static page without using a “User Text x” variable.

```
    /* OnNavigateError */
    function OnNavigateError ( session, params )
    {
        // Navigate to static error page
        var b = CETerm.Session( session ).Browser;
        b.Navigate( "file:///error.htm?" + params );
    }
```

The error page can access the failed URL parameters using normal techniques to re-attempt the navigation or decide on other error recovery.

```
    var params = document.location.search;
```

4.6 THE ONNAVIGATEREQUEST EVENT

The `OnNavigateRequest` event is fired before the browser begins a navigation. Normally, all navigation control should be done within the HTML of a page. In rare cases when the pages cannot be modified, this handler can be used to control navigations. This handler can cancel a navigation or request an alternative navigation.

Syntax

```
function OnNavigateRequest ( session, url )
```

session – index of browser session.
url – target URL of navigation request.

The return value of OnNavigateRequest is used to control the navigation. A return value of 0 will allow the navigation to continue, a value greater than 0 will cancel the navigation.

Example

This example shows how to control navigation.

```
/* OnNavigateRequest */
function OnNavigateRequest( session, url )
{
    // Prevent unwanted URL
    if (url.match( "forbidden.htm" )) return 1;

    // Require password
    if (url.match( "protected.htm" ))
    {
        var t = CETerm.TextInput;

        t.Title = "Administrator Login";
        t.Prompt = "Please enter your password:";
        t.PasswordMode = true;
        t.Input = "";  // Clear current password

        var s = t.GetInput();
        if (s !== 1 || t.Input !== "secret")
        {
            // Bad password, cancel
            return 1;
        }
        t.Input = ""; // Clear password
    }
```

```
        // Continue all other navigations
        return 0;
    }
```

4.7 THE ONNETCHECKFAILED EVENT

The `OnNetCheckFailed` event is fired if a “Network Check on Send” fails to detect the host system and the Network Check Action is “ida://IDA_SCRIPT_ON_NETCHECKFAILED”. Other Network Check Actions are possible, including direct naming of an error URL. See the User Manual for more information. Typically, this error handler will redirect the web browser to a “file:” URL on the device for error recovery.

Syntax

```
function OnNetCheckFailed ( session, pendingURL )
```

session – index of browser session attempting navigation.
pendingURL – pending URL for navigation.

The pendingURL is the destination that the user requested but which has been deferred because the host was not contacted. The event handler can re-try the navigation.

Example

This example is nearly identical to the `OnNavigateError` handler except that there is no error number in the pendingURL. This handler
shows how to redirect a Windows CE web browser to a static URL on the device.

```
    /* OnNetCheckFailed */
    function OnNetCheckFailed ( session, pendingURL )
    {
        // Save pendingURL in text 3x where x is session index
        // This is required by CE 5.0 devices which do not pass
        // parameters to a "file:" URL.
        CETerm.SetProperty( "app.usertext.3" + session, pendingURL );

        // Navigate to static error page
        var b = CETerm.Session( session ).Browser;
        b.Navigate( "file:///error.htm" );
    }
```

See the `OnNavigateError` example above for additional details.

4.8 THE ONPROGRAMEXIT EVENT

The `OnProgramExit` event is fired just before CETerm exits. This handler can abort the exit to keep CETerm running.

Syntax

```
function OnProgramExit ( )
```

This handler has no arguments. The function returns 0 to continue with the exit or 1 to abort the exit.

Example

This example prevents the CETerm exit if any session is connected.

```
/* OnProgramExit */
function OnProgramExit ( )
{
    // Don't exit if any session is connected
    for (var i=1; i<=CETerm.MaxSession; ++i)
    {
        if (CETerm.Session(i).IsConnected)
        {
            // Switch to first connected session
            CETerm.PostIDA( "IDA_SESSION_S" + i, 0 );

            // Abort exit
            return 1;
        }
    }

    // OK to exit
    return 0;
}
```

4.9 THE ONPROGRAMSTART EVENT

The `OnProgramStart` event is fired just before CETerm starts processing user input. All command line arguments are processed and auto-connect sessions are connected before this event.

Syntax

```
function OnProgramStart ( )
```

This handler has no arguments and any return value is ignored.

Example

This example makes sure Session 3 is active when CETerm starts.

```
/* OnProgramStart */
function OnProgramStart ( )
{
    // Allways switch to Session 3
    CETerm.PostIDA( "IDA_SESSION_S3", 0 );
}
```

4.10 THE ONSERIALPORTEVENT EVENT

The `OnSerialPortEvent` event is fired when a serial port changes state. The state change may be due to the arrival of data or due to the change of a signal line state. See Chapter 5 for details about `OnSerialPortEvent`.

Syntax

```
function OnSerialPortEvent ( portIndex, eventMask )
```

portIndex – index of serial port object signaling the event
eventMask – mask indicating event(s) that occurred

Example

This example shows the skeleton of the handler. See Chapter 5 for details.

```
/* OnSerialPortEvent */
function OnSerialPortEvent ( portIndex, eventMask )
{
    var EV_RXCHAR = 0x0001;   // Any Character received
    if (portIndex === 3 && (eventMask & EV_RXCHAR))
    {
        // Read data from port 3
        MyReadData( portIndex );
    }
}
```

4.11 THE ONSESSIONCONNECT EVENT

The `OnSessionConnect` event is fired when a session initially connects to the specified host. The handler can be used to initiate an automated login using the “expect” script and “ExpectMonitor” class.

Syntax

```
function OnSessionConnect ( session )
```

session – index of session which connected.

Example

An example using OnSessionConnect to start the automated login was shown above in Section 2.6 and is repeated below. Please refer to Section 2.6 for details. The “expect” script is discussed in Section 5.1.

```
/* OnSessionConnect */
function OnSessionConnect ( session )
{
    // Set login information
    var myusername = "joeuser";
    var mypassword = "secret";

    var waittime = 8000;    // Milliseconds waiting for each text

    // Only login session 1
    if (session == 1)
    {
        // Look for "login" then "password"
        expect( session, waittime, "Login", myusername + "\r",
                                   "Password", mypassword + "\r" );
    }
}
```

4.12 THE ONSESSIONDISCONNECT EVENT

The `OnSessionDisconnect` event is fired when a session is disconnected by a user action. The handler can be used to switch to another session, exit, or perform other cleanup tasks.

Syntax

```
function OnSessionDisconnect ( session )
```

session – index of session which was disconnected by user.

Example

This example will switch to the next connected session when the current session is disconnected. If there are no other connected sessions, then CETerm will exit.

```
/* OnSessionDisconnect */
function OnSessionDisconnect ( session )
{
    // Switch to next connected session
    CETerm.SendIDA( "IDA_SESSION_NEXTLIVE", 0 );

    if (CETerm.ActiveSession == session)
    {
        // Still on current session, no others connected.
        CETerm.PostIDA( "IDA_PROGRAM_EXIT", 0 );
    }
}
```

4.13 THE ONSESSIONDISCONNECTED EVENT

The `OnSessionDisconnected` event is fired when a terminal emulation (TE) session is disconnected by the remote host. The handler can be used to attempt to reconnect to the host or perform other cleanup tasks.

Syntax

```
function OnSessionDisconnected ( session )
```

session – index of session which was disconnected by remote host.

Example

This example will check for RF coverage and attempt to reconnect if RF is detected.

```
/* OnSessionDisconnected */
function OnSessionDisconnected ( session )
```

```
{
    // Check RF status
    var status = CETerm.GetProperty ( "device.rf.status" );
    if (status <= 0)
    {
        OS.Alert( "No RF signal detected.\n" +
                  "Return to RF coverage and reconnect." );
        return;
    }
    // Attempt to reconnect to host
    CETerm.PostIDA( "IDA_SESSION_CONNECT", session );
}
```

4.14 THE ONSESSIONRECEIVE EVENT

The `OnSessionReceive` event is fired when a terminal emulation session receives data from the connected host. The handler can be used to detect screen content such as an error message and perform a desired action.

Syntax

```
function OnSessionReceive ( session, count )
```

session – index of session which received data.
count – count of bytes received.

Example

This example will check the screen content on line 24 looking for an error message. If found, the error is displayed as a popup message.

```
/* OnSessionReceive */
function OnSessionReceive ( session, count )
{
    // Get line of text
    var s = CETerm.Session( session ).Screen;
    var line = s.GetTextLine( 24 );

    // Do a regular expression case-insensitive match
    if (line.match( /error/i ))
    {
        OS.Alert( "Error: " + line );
    }
}
```

4.15 THE ONSESSIONSWITCH EVENT

The `OnSessionSwitch` event is fired when the active session changes. The handler can be used to perform a session specific action.

Syntax

```
function OnSessionSwitch ( session, previousSession )
```

session – index of session which became active.
previousSession – index of session which was previously active.

Example

This example will reposition the battery information item depending on which browser session is active.

```
/* OnSessionSwitch */
function OnSessionSwitch ( session, previousSession )
{
    var b = CETerm.Session( session ).Browser;

    if (b.DocLoaded)
    {
        var x = (session == 1) ? 195 : 10;
        var y = (session == 1) ? 10 : 100;

        b.AddMetaItem( "Battery", "x=" + x );
        b.AddMetaItem( "Battery", "y=" + y );

        CETerm.PostIDA( "IDA_INFO_REFRESH", 0 );
    }
}
```

4.16 THE ONSTYLUSDOWN EVENT

The `OnStylusDown` event is fired when the user taps a terminal emulation screen with a stylus or finger. This event is only fired if the tap does not activate a standard “touch” feature. All touch features can be disabled in the CETerm configuration. This handler can be used to activate user-defined hot-spots.

Syntax

```
function OnStylusDown ( session, row, column )
```

session – index of currently active session
row – row of screen tap
column – column of screen tap.

Example

Several samples for OnStylusDown were given in Section 2.7. Following is an example that starts a barcode scan if the row contains the word “scan”. Not all hardware devices support a scan trigger by script. If tapping on an IBM screen, you must tap on an input field in the row, or the focus will not be in an input field when the scan is sent to the session. Of course the “OnBarcodeRead” handler could be used to force the scanned data into a preferred input field.

```
/* OnStylusDown */
function OnStylusDown ( session, row, column )
{
    var screen = CETerm.Session( session ).Screen;

   // Get row of text
    var text = screen.GetTextLine( row );

    // Look for "scan" as case-insensitive match
    if (text.match( /scan/i ))
    {
        CETerm.PostIDA( "IDA_SCAN_TRIGGER", 0 );
    }
}
```

4.17 THE ONTRIGGEREVENT EVENT

The `OnTriggerEvent` event is fired when the hardware trigger state changes.

Syntax

```
function OnTriggerEvent ( flags, id )
```

flags – flags describing state change
id – trigger id

Example

This example activates a corresponding trigger handling function in the current browser session.

```
/* OnTriggerEvent */
function OnTriggerEvent( flags, id )
{
    var index = CETerm.ActiveSession;
    var Session = CETerm.Session(index);

    if (Session.Browser.Document != null)
    {
        // Current session is browser session.
        // Hand off processing to web page.
        var script = "OnTriggerEvent("+ flags + "," + id + ");";

        Session.Browser.RunScript( script );
    }
}
```

4.18 THE ONVTCOMMAND EVENT

The `OnVTCommand` event is fired when a special command format is received by a VT emulation session. The command includes a variable number of arguments that depends on the received command. The handler script may perform any desired actions. The screen text may contain additional information used by the handler.

Syntax

```
function OnVTCommand ( session, command, arg1, arg2, arg3, arg4 )
```

session – index of session receiving the command
command – activating command, “ESCBangS”, “ESCTilda”, “APC”,
“OSC”, “PM”, “PU1”, “PU2”
arg1 – command argument. Terminating character for ESCTilda. Up to 16 arguments with values which range between 0 and 99 for ESCBangS. Unspecified ESCBangS arguments have the value -99. String with content for other commands.

Example

This example shows how to process the various possible arguments.

```
/* OnVTCommand */
function OnVTCommand( session, command, arg1, arg2, arg3, arg4 )
{
    if (command === "PM")
    {
        // Format: PM text ST  or  ESC ^ text ESC \
        // arg1 contains a text string
        // Other args are undefined
        if (arg1.charAt(0) === "1")
        {
            // Reply with special text
            CETerm.SendText( "TDT;001;038\n",  session );
        }
    }
    else if (command === "ESCBangS" )
    {
        // Format: ESC ! 1;2;3;4;5;6;7;8;9;0;1;2;3;4;5 S
        // Format (only two specified args): ESC ! 1;2 S
        // 16 arguments are defined
        // Arguments unspecified in the data stream have value -99
        // You can put more "argX" arguments in the function
        // definition or use the special JavaScript "arguments[]"
        // array to access the values:
        // arg1 === arguments[2]
        // arg2 === arguments[3]
        // ...
        // arg16 === arguments[17]
        // Process command
    }
    else if (command === "ESCTilda" )
    {
        // Format: ESC ~ E or ESC ~ S
        // arg1 is 69 (E) or 83 (S)
        // Process command
    }
}
```

4.19 THE ONWAKEUP EVENT

The `OnWakeup` event is fired when the device resumes after suspending. The handler can be used to perform any action, such as waiting for RF coverage or switching to a specific session.

Syntax

```
function OnWakeup ( )
```

Example

This example will wait for RF coverage to resume and sound a tone when it is available. While waiting, a “tic” sound will be made periodically to indicate the check. This sample is more complex than needed, but it illustrates how to use global variables and timers to periodically check state.

```
/* RFSoundOnConnect */

// Global control variables
var RFWakeupSoundTimerID = 0;
var RFWakeupSoundContinue = 0;
var RFWakeupSoundInterval = 200;    // milliseconds
var RFWakeupSoundCountMaximum = 50; // 50*200 = 10 seconds
var RFWakeupSoundCount = 0;


function OnWakeup ()
{
  // Start with wakeup event
  RFWakeupSoundStart();
}

// Function to start RF check
function RFWakeupSoundStart()
{
  if (!RFWakeupSoundContinue)
  {
    RFWakeupSoundContinue = 1;
    if (RFWakeupSoundTimerID != 0)
    {
      // Stop and clear any previous check
      CETerm.ClearTimeout( RFWakeupSoundTimerID );
      RFWakeupSoundTimerID = 0;
    }
    RFWakeupSoundCount = 0;

    // Schedule first RF check
    RFWakeupSoundTimerID = CETerm.SetTimeout(
                             "RFWakeupSoundOnTimer();",
                             RFWakeupSoundInterval );
  }
}

// Function to check RF and notify user
function RFWakeupSoundOnTimer()
{
  RFWakeupSoundTimerID = 0;
  RFWakeupSoundCount++;

  // Get and check info
  var rfStatus = CETerm.GetProperty( "device.rf.status" );
  if (rfStatus > 0)
  {
    // RF detected
    // Delayed playsound, increase delay for WEP if needed
    CETerm.SetTimeout( "RFWakeupSoundPlay();", 100 );
    RFWakeupSoundContinue = 0;
  }
  else if (RFWakeupSoundCount > RFWakeupSoundCountMaximum)
  {
    // Failed to get RF, show failure message.
    OS.Alert( "Failed to detect RF signal.\n" +
              "Return to coverage area." );
    RFWakeupSoundContinue = 0;
  }

  if (RFWakeupSoundContinue)
  {
    if (!(RFWakeupSoundCount % 5))
    {
      // Play "tick" sound while check is running.
      CETerm.PlaySound( "MenuPop" );
    }

    // Schedule next RF check
    RFWakeupSoundTimerID = CETerm.SetTimeout(
                             "RFWakeupSoundOnTimer();",
                             RFWakeupSoundInterval );
  }
}


function RFWakeupSoundPlay()
{
  // Select any wave file on device for notification.
  CETerm.PlaySound( "infbeg" );
}
```

5.0 Scripting Techniques and Tips

This section describes ways that scripting can extend the capabilities of CETerm. Tips for script development are also presented.

5.1 EXPECT AND EXPECTMONITOR FOR AUTOMATING TASKS

The “expect” script and “ExpectMonitor” class provide a general purpose “prompt-and-response” tool. Using “expect” for automated login was described in Section 2.6. Here we provide the complete listing of the scripts and discuss other options for use.

5.1.1 Expect Script

The “expect” script illustrates a couple of powerful JavaScript constructs. Even though the expect function has 4 defined arguments in the function declaration, it is possible to pass an unlimited number of arguments. All arguments are accessible through the special “arguments” variable. This script also shows the object-oriented aspects of JavaScript by creating a new ExpectMonitor class.

```
/* expect */
//
// This script will "expect" a text prompt on the screen and
// respond with text or action.
//
// Syntax: expect( session, timeout,
//                 expectedText, response
//                 [,expectedText2, response2] )
//
// session is the session index
// timeout is the wait interval for each text in milliseconds
// expectedText can be a string or regular expression
// Response can be a text response or a function

function expect( session, timeout, expectedText, response )
{
    // Build array from arguments
    // This technique will accumulate any
    // number of expect/response pairs
    var TargetResponseArray = [];
    for (var i=2; i < arguments.length; i++)
    {
        TargetResponseArray.push( arguments[i] );
    }

    // Create an ExpectMonitor class that manages the actions
    var EM = new ExpectMonitor ( session, timeout,
                                 TargetResponseArray );

    // Set optional ExpectMonitor behaviors
    //EM.silent = true;
    //EM.OnDone = function (success) { OS.Alert( "Done." ) };

    // Start check
    EM.Start();
}
```

5.1.2 ExpectMonitor Class

The “ExpectMonitor” class illustrates the use of a prototype in JavaScript. This class also manages all instances of itself to restrict the number of objects that can be created.

```
/* ExpectMonitor */
//
// ExpectMonitor class
//
// The ExpectMonitor class manages the expect/action
// sequence for a session.
// Only one ExpectMonitor is allowed per session.
//


function ExpectMonitor ( session, timeout, targetactions )
{
    // Validate session
    if (session < 1 || session > 4)
    {
        return null;
    }

    this.session = session;
    this.timeout = timeout;
    this.args = targetactions;

    this.state = 0;
    this.timer = null;
    this.checkCount = 0;
    this.totalCheckCount = 0;
    this.maxCheckCount = this.timeout / this.checkDelta;

    // Abort any existing object
    if (ExpectMonitor.Instances[this.session] != null)
    {
        ExpectMonitor.Instances[this.session].Abort();
    }

    // Record this instance in the global array
    ExpectMonitor.Instances[this.session] = this;
}
```

```
function ExpectMonitor_Check()
{
    // Clear timer id
    this.timer = null;

    // If something to check for, check it.
    var target = this.args[this.state];

    if (target != null)
    {
        // Get all screen text
        var screenText =
           CETerm.Session(this.session).Screen.GetText (1,1,-1,-1);

        if (screenText != null && screenText.match( target ))
        {
            // Found match
            var action = this.args[this.state + 1];
            this.checkCount = 0;

            if (action != null)
            {
                // Check action
                if (typeof action == "function")
                {
                    // Run function action
                    // Pass session number as argument
                    action( this.session );
                }
                else if (typeof action == "string")
                {
                    // Send text to session
                    CETerm.SendText( action, this.session );
                }
                else if (!this.silent)
                {
                    OS.Alert("Unknown action type for expect.");
                }
            }

            // Check if another match expected
            this.state +=2;
            target = this.args[this.state];

            if (target != null)
            {
                // Schedule next check
                this.Schedule();
            }
            else
            {
                // Done with this expect.
                // Run any post-execution actions
                if (typeof this.OnDone == "function")
                {
                    this.OnDone( true );
                }
            }
        }
        else
        {
            // No match, schedule again
            if (this.checkCount++ < this.maxCheckCount)
            {
                this.Schedule();
            }
            else
            {
                if (!this.silent)
                {
                    OS.Alert( "Expect failed to find text \"" +
                              target + "\"" );
                }
                if (typeof this.OnDone == "function")
                {
                    // Done but failed
                    this.OnDone( false );
                }
            }
        }
    }
}


function ExpectMonitor_Schedule()
{
    // Schedule next check
    var script = "ExpectMonitor.Instances[" +
                    this.session + "].Check()";
    this.timer = CETerm.SetTimeout( script, this.checkDelta );
}


function ExpectMonitor_Start()
{
    // Cleanup first in case restarted
    this.Abort();

    // Initialize state
    this.state = 0;
    this.checkCount = 0;

    this.Check();
}
```

```
function ExpectMonitor_Abort()
{
    // Stop any timer
    if (this.timer != null)
    {
        CETerm.ClearTimeout( this.timer );
        this.timer = null;
    }

    // Set state to beyond reasonable range
    this.state = 1000;
}


// Method definitions
ExpectMonitor.prototype.Check = ExpectMonitor_Check;
ExpectMonitor.prototype.Schedule = ExpectMonitor_Schedule;
ExpectMonitor.prototype.Start = ExpectMonitor_Start;
ExpectMonitor.prototype.Abort = ExpectMonitor_Abort;

ExpectMonitor.prototype.OnDone = null;


// Check every 200 milliseconds
ExpectMonitor.prototype.checkDelta = 200;
// About 10 seconds for each text check
ExpectMonitor.prototype.maxCheckCount = 50;
// Allow messages
ExpectMonitor.prototype.silent = false;


// Class statics
ExpectMonitor.Instances = [];
```

5.1.3 Automating Tasks with Expect

Any routine prompt-and-response task can be automated with “expect”. Examples may be navigating through a hierarchy of menus or closing an order for shipping. In any case, you identify text to find on the screen and the user input to take you to the next screen. Here is a simple menu traversal:

```
// Traverse menu
expect( CETerm.ActiveSession, 8000,
        "3. Applications", "3\r",
        "2. Inventory", "2\r",
        "2. Put Back", "2\r" );
```

This script can be entered into any script slot and bound to a key combination for activation. You must also load the "expect" and "ExpectMonitor" in a script slot which is marked "Load at Startup" so that the functions are available for use.

5.2 PRESENTING VISUAL FEEDBACK DURING SCRIPT EXECUTION

The Message object can be displayed during script execution when you want to provide a visual indication of script progress. The Message object is asynchronous and a script can continue running while it is displayed. This is unlike the OS.Alert() message which stops script execution and requires user confirmation. There is only one Message object within CETerm and you can change the Message properties within any script.

WARNING: You must exercise caution when using the Message box to avoid leaving it visible after a script is done. You may want to provide a cleanup script that can be activated by the user to be sure the message is hidden.

Following is an example of using the Message box. This message will display itself for 5 seconds and then disappear.

```
/* Show message for 5 seconds */
var m = CETerm.Message;
m.Text = "Processing data, please wait.";
m.Timeout = 5;
m.AbortButtonVisible = true; // does nothing because script exits
m.Progress = 0;
m.ProgressRunning = true;
m.ProgressVisible = true;
m.ProgressRate = 20;
m.IsVisible = true;
```

You may want to update the progress bar directly while processing data. Here is an example.

```
/* Update progress and message during processing */
var m = CETerm.Message;
m.Text = "Processing data, please wait.";
m.Timeout = 0;
m.AbortButtonVisible = false;
m.ProgressRunning = false;

// Do some work
```

```
m.Progress = 0;
m.IsVisible = true;
OS.Sleep( 2000 );  // Simulate work delay

// Update
m.Progress = 20;
m.Text = "Finding addresses, please wait.";
OS.Sleep( 2000 );  // Simulate work delay

// Update
m.Progress = 50;
m.Text = "Sorting addresses, please wait.";
OS.Sleep( 2000 );  // Simulate work delay

// Update
m.Progress = 90;
m.Text = "Almost done, please wait.";
OS.Sleep( 2000 );  // Simulate work delay

// Done
m.IsVisible = false;
```

5.3 GETTING USER INPUT TO A SCRIPT

The TextInput object can get user input for a script. Here is an example for getting a password.

```
/* Get password from user */
var t = CETerm.TextInput;

t.Title = "Warehouse Management";
t.Prompt = "Please enter your password:";
t.PasswordMode = true;
t.Input = "";  // Clear current password

var s = t.GetInput();
if (s == 1)
{
   OS.Alert( "Password is " + t.Input );
   t.Input = "";  // Clear password
}
else
{
   OS.Alert( "Failed to get password." );
}
```

5.4 RUNNING AN EXTERNAL PROGRAM

It is possible to start an external program from the CETerm script engine. You can wait for the program to finish or allow it to run independently. Often you will run a program then return to CETerm when it exits. The `Process` object allows you to manage running processes. The `Event` object can be used to schedule a script to run when a process exits.

Here is an example to start the stylus calibration. Note that the arguments depend on whether your device is Window CE or Windows Mobile.

```
/* Stylus Calibration */

// TODO: Uncomment the lines for your device

// For Windows CE 5.0 devices
OS.Process.ExecuteAction ( "\\Windows\\ctlpnl.exe",
                           "cplmain.cpl,9,1", "open" );

// For Windows Mobile 5 devices
//OS.Process.ExecuteAction ( "\\Windows\\ctlpnl.exe",
//                           "cplmain.cpl,7,0", "open" );
```

5.5 USING TIMERS TO RUN SCRIPTS

Script execution timers are useful for several tasks. They can be used to:

1. Defer an action which is not possible in an event handler.
2. Perform an action periodically.
3. Provide an asynchronous script execution.
4. Split up a long running task.

We have already shown how the timer is used with the ExpectMonitor class and task automation in Section 5.1. Event handlers should be limited to a small amount of processing. If more processing is needed, it is best to schedule that processing with SetTimeout() and allow the event handler to exit.

The following example will save data from memory to a flash file whenever a particular URL is loaded.

```
/* OnDocumentDone */
function OnDocumentDone ( session )
{
    var b = CETerm.Session( session ).Browser;

    if (b.Document.URL.match( /InventorySave/ ))
```

```
        {
            // Resume online inventory, and save cached
            // data to file in background.
            CETerm.SetTimeout( "BackgroundSave(" + session + ");", 10 );
        }
    }

    /* BackgroundSave */
    function BackgroundSave( session )
    {
        var d = new ActiveXObject( "Microsoft.XMLDOM" );
        d.loadXML(
          "<?xml version=\"1.0\"?><Books>" +
          "<Book QTY=\"10\"><Title>Beginning XML</Title></Book>" +
          "<Book QTY=\"2\"><Title>Mastering XML</Title></Book>" +
          "</Books>");

        if (!OS.File.Write( "\\FlashDisk\\inventory.xml", d.xml ))
        {
            OS.Alert( "Failed to save inventory." );
        }
    }
```

5.6 ACCESSING A FILE

The File automation object provides basic access to the Windows CE filesystem. It supports whole-file read and write, but does not support the concept of an "open" file with piecewise read or write. You can also create and delete file directories.

This example shows how to append to an existing file by using a combination of read and write. The new File.Append() method should now be used to append data to files but this example still illustrate how to use the File object.

```
    /* AppendToFile */
    function AppendToFile( filename, addedContent )
    {
        var status = false;
        var F = OS.File;

        // Check if file exists
        var attributes = F.GetAttributes( filename );
        if (attributes != 0xFFFFFFFF)
        {
            var content = F.Read( filename );
            status = F.Write( filename, content + addedContent );
        }
        else
        {
            status = F.Write( filename, addedContent );
        }

        return status;
    }
```

5.7 ACCESSING THE REGISTRY

The registry on a Windows CE device is a form of database which contains most of the device configuration. The `Registry` automation object allows you to read, write and delete registry keys and values.

WARNING: Altering the registry can make your device unusable. Be sure you understand the effect of changing values and accept the responsibility.

The registry has a hierarchical structure. The "keys" are similar to file folders and the "values" inside a key are similar to files in a folder. Several `Registry` methods require a "fully qualified" value name which contains the full key hierarchy, begins with a "root" key, and ends with the value name. This fully qualified value name is similar to a file name with the full path.

The following example creates a new key and value and confirms that it can be read.

```
    /* NewRegistryDWORD */
    function NewRegistryDWORD( keyname, valuename, valuedata )
    {
        var status = false;
        var R = OS.Registry;

        // Check if file exists
        var fullyQualifiedKey = "HKEY_LOCAL_MACHINE\\" +
                        keyname + "\\" + valuename;

        if (!R.WriteValue( fullyQualifiedKey, valuedata, "REG_DWORD" ))
        {
            // Check if can read value
            var readdata = R.ReadValue( fullyQualifiedKey );
            if (readdata == valuedata)
            {
                status = true;
            }
        }

        if (!status)
        {
            OS.Alert( "Failed to confirm write of " +
                      fullyQualifiedKey );
        }

        return status;
    }
```

NewRegistryDWORD may be used as follows.

```
    // Write a new value
    NewRegistryDWORD( "SOFTWARE\\Naurtech\\Test", "TestDword", 510 );
```

5.8 CONTROLLING A SERIAL PORT FROM CETERM

This section describes how to control serial ports with scripting in CETerm. Both real (e.g., "COM1:") and virtual (e.g., "BSP1:") serial ports can be fully controlled and accessed from emulation or browser sessions.

Serial ports can be opened for read-access, write-access, or both. All serial port settings can be configured and events can be generated when data is available or when signal lines change state.

The `SerialPort` object can be used to integrate any serial device into CETerm; such as a tethered scanner, scale, printer, Bluetooth scanner, or RFID reader. As with all CETerm scripting features, we provide as much direct access to the hardware as possible while hiding un-needed complexity. The `SerialPort` object usage can be complex and the developer will need a good programming foundation with event-driven concepts. Much of the `SerialPort` object corresponds directly to the Windows Win32 serial port APIs. General information can be found by searching msdn.microsoft.com with the keywords "basic serial communication".

5.8.1 SerialPort Objects

CETerm provides access to ten (10) `SerialPort` objects. By default, these correspond to "COM0:" through "COM9:", however, any `SerialPort` object can be configured to control any named port, such as "BSP1:" for a virtual Bluetooth port. The `SerialPort` objects can be used from both the CETerm script engine and the browser scripting environment.

When a `SerialPort` object is first accessed within CETerm, it begins with all default settings and will maintain all updated settings while CETerm is running.

The `SerialPort` objects are obtained from the root `Device` object by specifying the desired index:

```
     var index = 0;
     var sp = Device.SerialPort( index );
```

where `sp` is the reference to the `SerialPort` object. It is a good practice to use a local variable to hold the reference inside a function when multiple port operations must be performed.

5.8.2 Setting the PortName

By default, the `SerialPort` objects correspond to the “COMx:” ports. If you plan to control “COM1:” then you should use `Device.SerialPort(1)`. In a few cases, you may need to control a non-COM port, such as a virtual serial port “BSP1:” for a Bluetooth device. In this case, you can use any `SerialPort` object that is not being used for a COM port and set the `PortName` as needed.

```
   var bluetoothPortIndex = 5;
   var bluetoothPortName = "BSP1:";
   var sp = Device.SerialPort( bluetoothPortIndex );
   sp.PortName = bluetoothPortName;
```

It is also possible to specify COM ports with numbers greater than 9. Following the Windows convention, prepend the string "$device\\" to the name. The double backslash is required in JavaScript literal strings to specify a single backslash character. For example, to open COM123, use the name "$device\\COM123:". You must set the name before opening the port.

5.8.3 Configuring SerialPort Properties

All serial port settings can be controlled through `SerialPort` object properties. Please refer to Section 3.15 for a complete list of properties. In general, you will set needed properties before opening the port for the first time, but most properties can be changed at any time.

Some properties are intuitive and control well known settings such as the baud rate. Other properties, such as timeouts and “handshaking” can be more complex and confusing. It can be helpful to search for information at msdn.microsoft.com with the keywords “serial communications reference ce” to learn additional details about Windows CE serial port control and behavior.

5.8.3.1 Configuring SerialPort Timeouts

When reading from the serial port, the default timeouts prevent blocking of the read operation. Most reads should be performed in response to an EV_RXCHAR event indicating that data is available from the attached device. Often however, some timeout is needed to allow a complete data message to be received. In these cases, we recommend a small `ReadTotalTimeoutConstant` and perhaps a small `ReadTotalTimeoutMultiplier`. Beware of a read request with a large maxCount parameter because this can result in a large total timeout if `ReadTotalTimeoutMultiplier` is non-zero.

```
var portIndex = 5;
var sp = Device.SerialPort( portIndex );
sp.ReadTotalTimeoutConstant = 100;
sp.ReadTotalTimeoutMultiplier = 0;
sp.ReadIntervalTimeout = 0;
```

You should review the Microsoft documentation mentioned above and may need to experiment with timeouts to configure the behavior that works best with your peripheral and software architecture. For example, if you do not seem to get a “complete message” during your read, you may need a larger timeout or may need to save the partial data and perform another read at a later time.

When writing to the serial port, the default timeouts wait for all the data to be written. In most cases this will be instantaneous because the data are placed in the output buffer. If flow-control is enabled and transmission is blocked, the buffer may fill and the write may block. You should use write timeouts if blocking is possible so that CETerm is not fully blocked waiting for the write to complete.

```
var portIndex = 5;
var sp = Device.SerialPort( portIndex );
sp.WriteTotalTimeoutConstant = 100;
sp.WriteTotalTimeoutMultiplier = 10;
```

5.8.3.2 Configuring Common SerialPort Properties

In addition to the timeout properties discussed above, there are a couple of other properties which must often be configured. These include the baud rate, flow control and the `EventMask`. Here is a sample function used to configure and open a port.

```
function OpenPort( portIndex )
{
    var sp = Device.SerialPort( portIndex );

    // Port configuration
    sp.EventMask = EV_RXCHAR | EV_DSR;

    sp.XOFFOutputFlowControl = false;
    sp.XOFFInputFlowControl = false;

    // See Appendix 4 for contants
    sp.BaudRate = CBR_9600;
    sp.DataBits = 8;
    sp.StopBits = ONESTOPBIT;
    sp.ParityMode = NOPARITY;

    // Set read timeouts
    sp.ReadTotalTimeoutConstant = 100;
    sp.ReadIntervalTimeout = 50;
    sp.ReadTotalTimeoutMultiplier = 10;

    // Set write timeouts
    sp.WriteTotalTimeoutConstant = 100;
    sp.WriteTotalTimeoutMultiplier = 0;

    // Open port
    return sp.Open( GENERIC_READ | GENERIC_WRITE );
}
```

5.8.4 Using WaitForEvent to Detect Data and State Changes

Section 4.0 describes why scripts cannot run continuously within CETerm. While a script is blocking on a `SerialPort.Read` command waiting for data, CETerm cannot respond to user input or perform other actions. Because of this, you must keep read timeouts short to maintain a responsive program. Although you could "poll" the serial port frequently to read newly arrived data, this is an inefficient technique. The better technique is to use the `SerialPort.WaitForEvent` method to run an event handler when data arrives or other states change.

To use `WaitForEvent`, you first configure the types of events you want to detect. These are set as flag values in the `EventMask` property. For example, to report events for the arrival of data and a change of the DSR state you would use

```
var portIndex = 5;
var sp = Device.SerialPort( portIndex );
sp.EventMask = EV_RXCHAR | EV_DSR;  // See Appendix 4
```

You must also define an `OnSerialPortEvent` handler that will be called when the event occurs. Here is a sample handler:

```
// Serial port event handler
function OnSerialPortEvent( portIndex, eventMask )
{
    if (portIndex === 5)
    {
        if (eventMask & EV_RXCHAR)
        {
            // Data is available, read and process
            MyReadAndProcess( portIndex );
        }

        if (eventMask & EV_DSR)
        {
            // DSR state changed.  Device entered sleep
            MyCloseAndReopen( portIndex );
        }
    }
    else if (portIndex === 3)
    {
        // Do something different for port 3
        DoPort3Actions( eventMask );
    }
}
```

This sample is just a template. It shows that there is only one `OnSerialPortEvent` handler for all serial ports, and that you must further direct the event to your own processing routines depending on the port signaling the event. The event handler may be very complex. Some rich examples are available on our website or through Naurtech Support.

After your handler is defined and the port opened, you call `SerialPort.WaitForEvent` when you are ready to handle events. This enables an event listener for the port. If the event is signaled by the port, the event handler `OnSerialPortEvent` is invoked. The event handler is invoked only once for each call of `WaitForEvent`, but the handler parameters may indicate multiple event conditions. Within the `OnSerialPortEvent` handler, or other helper routines, a common pattern is to schedule the next `SerialPort.WaitForEvent`. Although they share a common handler function, you must call `WaitForEvent` separately for each port, each time you want to enable events for that port.

To cancel an active event listener use `SerialPort.CancelWaitForEvent`.

5.8.5 Using Single Byte Reads

One pattern which often works well for handling serial port data is to read a single byte at a time and to accumulate data until a "complete message" can be processed. Your application and peripheral device will define what a "complete message" contains, but often messages are terminated by special characters such as ASCII ETX. When using single-byte reads, you can keep read timeouts short and optimize responsiveness.

To use this pattern, you would use WaitForEvent with the EV_RXCHAR event and process the data within your handler. Here is a template for a single byte read handler

```
// Serial port data handler
var responseData = [];
var responseState = "PRE_STX";

function MyReadAndProcess( portIndex )
{
    var c;
    var readTries = 4;
    var sp = Device.SerialPort( portIndex );

    while (responseState !== "DONE" &&
           readTries > 0)
    {
        c=sp.ReadByte();
        if (c < 0)
        {
            // Read error, try again
            OS.Sleep( 20 );
            --readTries;
            continue;
        }

        // Reset tries after successful read
        readTries = 4;

        switch(c)
        {
        case ASCII_STX:
            // Start of response.
            responseState = "DATA";
            responseData = [];
            break;

        case ASCII_ETX:
            // End of content
            responseState = "DONE";
            break;
        default:
            if (responseState === "DATA")
            {
                // Save character
                responseData.push( String.fromCharCode( c ) );
            }
            break;
        }
    }

    if (responseState === "DONE")
    {
        // Process message
        // Maybe schedule another data detection after processed.
        MyProcessMessage( responseData );
        responseState = "PRE_STX";
    }
    else
    {    // Schedule another data detection to complete message
         sp.WaitForEvent();
    }
}
```

5.9 WRITING EFFICIENT SCRIPTS

Good programming practices should be used when developing scripts for CETerm. In general, it is important to conserve memory, minimize script compilations, and limit execution times. Please refer to a JavaScript programming book for more information. We recommend “JavaScript: The Definitive Guide (5th Edition)” by David Flanagan. The following URL is an excellent starting point for in-depth details about JavaScript and good programming practices: http://javascript.crockford.com/.

5.9.1 Use Local Variables

Whenever possible, use local variables within functions and declare them with the var keyword, like this:

```
var status;
var message = "hello";
var i, j, k;
```

If you fail to use the var keyword, then JavaScript automatically creates a global variable with that name if it has not already been declared outside a function.

JavaScript uses “garbage collection” to reclaim memory no longer in use. Memory occupied by global variables may never be reclaimed, whereas local variable memory can be reclaimed after a function call completes. Because the JavaScript engine in CETerm is not reset frequently like a browser JavaScript engine, it is more likely that poor programming practices could exhaust memory.

5.9.2 Encapsulate Code in Functions

Whenever possible, put multiple script actions within a function. This should minimize compilations and make it easier to use local variables as described above. For example, the following actions could be in a script which is bound to a key-combination:

```
CETerm.SetProperty( "session1.scanner.upca.enabled", true );
CETerm.SetProperty( "session1.scanner.msi.enabled", false );
CETerm.SetProperty( "session1.scanner.pdf417.enabled", false );
CETerm.PlayTone( 8, 2000, 200 );
CETerm.PlayTone( 8, 1500, 200 );
CETerm.PostIDA( "IDA_SCAN_APPLYCONFIG", 0 );
```

Or, the actions could be in a function which is loaded with “Load at Startup”

```
function enableUPCA()
{
    CETerm.SetProperty( "session1.scanner.upca.enabled", true );
    CETerm.SetProperty( "session1.scanner.msi.enabled", false );
    CETerm.SetProperty( "session1.scanner.pdf417.enabled", false );
    CETerm.PlayTone( 8, 2000, 200 );
    CETerm.PlayTone( 8, 1500, 200 );
    CETerm.PostIDA( "IDA_SCAN_APPLYCONFIG", 0 );
{
```

and the function call, in a separate script, could be bound to the key-combination:

```
enableUPCA();
```

Using the later approach, the function is only compiled once, not each time the key is pressed. In general, separating the function definitions from the invocation is a good practice.

5.9.3 Limit Execution Time

Because the script engine acts like a “virtual user”, when a script is executing, CETerm will seem unresponsive. You cannot have a script running continuously. However, using events and timers, you can accomplish any task.

Do not disable the “Script Timeout” unless you are sure your script will not enter an infinite loop.

5.10 DEBUGGING SCRIPTS

All but the most trivial script will require some amount of debugging.

5.10.1 Show Script Errors

The first step is to enable “Show Script Errors”. This will enable a popup message for compilation and runtime errors. Compilation errors will usually be seen when new scripts are added or upon script engine startup. It may not be clear which script loaded at startup contains the error. In this case you may need to open the edit dialog for each script and tap the “Test/Load” button to identify the bad script.

The compilation error looks like this:

```
Microsoft JScript compilation error
[Line: 15 Col: 8] Expected ‘)’
        OS.Alert( message );
```

Notice that the line of script presented looks OK. In this case, the missing ‘)’ is on the previous line of script, but the error is detected as the compiler reaches column 8 of this line and encounters the ‘O’. Be sure to look around the indicated location to identify the source of the error.

A runtime error may be seen at startup if a script is performing some initialization, or it may be seen while using CETerm. It can be difficult to identify the source of the error if the script was fired by an event or timer. Most often, a runtime error can be prevented by “defensive coding” where you are sure to check the validity of arguments and object references.

The runtime error looks like this:

```
Microsoft JScript runtime error
```

```
[Line: 14 Col: 9] Object doesn't support this property
or method.
```

Unfortunately, the JScript engine does not return the source code line for a runtime error. You must manually examine your scripts at the specified location for a clue about the problem.

5.10.2 OS.Alert()

Because there is no JScript debugger on the Windows CE device, the tried-and-true debugging tool is "OS.Alert(message)". Experienced programmers will recognize this as the "write(6,100)", "printf" or "MessageBox" technique.

The basic idea is to sprinkle "OS.Alert()" calls through your code to track program flow and variable values. It can be tedious, but it's easy to do and easy to remove the OS.Alert() calls by preceding them with comment characters.

Alternatively, you can define a Debug() method and sprinkle it through your code. This makes it easier to enable or disable debugging.

```
var globalDebugLevel = 0;

function Debug( message )
{
    if (globalDebugLevel > 0)
    {
        OS.Alert( message );
    }
}
```

Appendix 1 - IDA Action Codes

Many IDA codes apply only to a Terminal Emulation session. Some IDA codes can only be used in restricted circumstances, such as IDA_URL.

Symbolic Name	**Friendly Name**	**Description**
IDA_BEL	Bell	
IDA_BS	Backspace	
IDA_HT	Horizontal Tab	
IDA_TAB	Tab	
IDA_LF	Linefeed	
IDA_VT	Vertical Tab	
IDA_FF	Form Feed	
IDA_CR	Carriage Return	
Printable ASCII		
IDA_SPACE	<Space>	
IDA_EXCLAMATION_MARK	!	
IDA_DOUBLE_QUOTE	"	
IDA_NUMBER_SIGN	#	
IDA_DOLLAR_SIGN	$	
IDA_PERCENT	%	
IDA_AMPERSAND	&	
IDA_SINGLE_QUOTE	'	
IDA_LEFT_PAREN	(	
IDA_RIGHT_PAREN	)	
IDA_ASTERISK	*	
IDA_PLUS	+	
IDA_COMMA	,	
IDA_HYPHEN	-	
IDA_PERIOD	.	
IDA_SLASH	/	
IDA_0	0	
IDA_1	1	
…	…	
IDA_9	9	
IDA_COLON	:	
IDA_SEMICOLON	;	
IDA_LESS_THAN	<	
IDA_EQUAL	=	
IDA_GREATER_THAN	>	
IDA_QUESTION_MARK	?	

Symbolic Name	**Friendly Name**	**Description**
IDA_AT	@	
IDA_A	A	
IDA_B	B	
…	…	
IDA_Z	Z	
IDA_LEFT_BRACKET	[	
IDA_BACKSLASH	\	
IDA_RIGHT_BRACKET	]	
IDA_CARET	^	
IDA_UNDERSCORE	_	
IDA_BACKTICK	`	
IDA_a	a	
IDA_b	b	
…	…	
IDA_z	z	
IDA_LEFT_BRACE	{	
IDA_PIPE	|	
IDA_RIGHT_BRACE	}	
IDA_TILDE	~	
IDA_DEL	DEL	
C1 ASCII Controls		
IDA_IND	Index	
IDA_NEL	Next Line	
IDA_HTS	Horiz Tab Set	
IDA_RI	Reverse Index	
IDA_SS2	Single Shift 2	
IDA_SS3	Single Shift 3	
IDA_DCS	Device Ctrl Str	
IDA_PU1	Private Use One	
IDA_PU2	Private Use Two	
IDA_CSI	Ctrl Seq Intro	
IDA_ST	String Term	
IDA_OSC	OS Command	
IDA_PM	Private Msg	
IDA_APC	App Prog Cmd	
Internal Actions (TE only)		
IDA_UPDATE_CURSOR	Update Cursor	

Symbolic Name	**Friendly Name**	**Description**
IDA_INHIBIT_UPDATE	Inhibit Update	Don't update display
IDA_UNINHIBIT_UPDATE	Uninhibit Update	Allow display update
IDA_UPDATE	Update	Force display update
IDA_INHIBIT_SEND	Inhibit Send	VT buffer characters
IDA_UNINHIBIT_SEND	Uninhibit Send	VT stop buffering
IDA_SEND_PENDING	Send Pending Chars	VT send buffered chars
Program Actions		
IDA_PROGRAM_ABOUT	Program About	Display About dialog
IDA_PROGRAM_EXIT	Program Exit	Exit program
IDA_PROGRAM_EXITSILENT	Program Exit Silent	Exit program silently
IDA_PROGRAM_HELP	Program Help	Display Help
IDA_SUSPEND_DEVICE	Suspend Device	Enter suspend state
IDA_BLUETOOTH_DISCOVERY	Bluetooth Discovery	Start discovery
IDA_WARMBOOT	Warm Boot	Warm boot device
IDA_COLDBOOT	Cold Boot	Cold boot device
IDA_MENU_TOPBOTTOM	Menu Top/Bot	Toggle menu location
IDA_MENU_TOGGLEHIDE	Menu Toggle	Toggle menu visibility
IDA_TOOLBAR_TOGGLE	ToolBar Toggle	Toggle toolbar visibility
IDA_START_TOGGLEHIDE	Start Menu Toggle	Toggle Start visibility
IDA_MENUBAR_TOGGLEHIDE	MenuBar Toggle	Toggle menubar visibility
IDA_SESSION_TOGGLECON	Connect/Disconnect	Toggle session connection
IDA_SESSION_CONFIGURE	Configure	Configure session
IDA_SESSION_CONNECT	Connect	Connect session
IDA_SESSION_DISCONNECT	Disconnect	Disconnect session
IDA_SESSION_NEXT_LIVE	Next Live Session	Switch to next live session
IDA_SESSION_PASSWORD	Password	Session password dialog
IDA_SESSION_PREV	Prev Session	Switch to previous session
IDA_SESSION_NEXT	Next Session	Switch to next session
IDA_SESSION_DISCON_ALL	Disconnect All	Disconnect all sessions
IDA_SESSION_S1	Session 1	Switch to session 1
IDA_SESSION_S2	Session 2	Switch to session 2
IDA_SESSION_S3	Session 3	Switch to session 3

Symbolic Name	Friendly Name	Description
IDA_SESSION_S4	Session 4	Switch to session 4
IDA_TOOLBAND_HIDE	Hide ToolBar	Hide full Toolbar
IDA_TOOLBAND_TOGGLEHIDE	Toggle ToolBar	Toggle Toolbar visibility
IDA_KEYBAR_HIDE	Hide KeyBar	Hide KeyBar
IDA_KEYBAR_TOGGLEHIDE	KeyBar Toggle	Toggle KeyBar visibility
IDA_KEYBAR_LEFT	Prev KeyBar	Switch to previous KeyBar
IDA_KEYBAR_RIGHT	Next KeyBar	Switch to next KeyBar
IDA_KEYBAR_SEPARATOR	--Separator--	Separator for KeyBar
IDA_KEYBAR_NONE	(Empty)	No action placeholder
IDA_HSCROLL_HIDE	HScroll Hide	
IDA_HSCROLL_VISIBLE	HScroll Show	
IDA_HSCROLL_TOGGLEHIDE	HScroll Toggle	
IDA_HSCROLL_PLUSON	HScroll Right One	
IDA_HSCROLL_MINUSONE	HScroll Left One	
IDA_HSCROLL_PLUSHALF	HScroll Right Page	
IDA_HSCROLL_MINUSHALF	HScroll Left Page	
IDA_HSCROLL_PLUSEND	HScroll Right End	
IDA_HSCROLL_MINUSEND	HScroll Left End	
IDA_VSCROLL_HIDE	VScroll Hide	
IDA_VSCROLL_VISIBLE	VScroll Show	
IDA_VSCROLL_TOGGLEHIDE	VScroll Toggle	
IDA_VSCROLL_PLUSONE	VScroll Up One	
IDA_VSCROLL_MINUSONE	VScroll Down One	
IDA_VSCROLL_PLUSHALF	VScroll Up Page	
IDA_VSCROLL_MINUSHALF	VScroll Down Page	
IDA_VSCROLL_PLUSEND	VScroll Up End	
IDA_VSCROLL_MINUSEND	VScroll Down End	
IDA_FONT_PLUS	Font Inc	Increase font size
IDA_FONT_MINUS	Font Dec	Decrease font size
IDA_TOGGLE_FONT_BOLD	Font Bold	
IDA_SMARTPAD_OPEN	SmartPad Show	
IDA_SMARTPAD_CLOSE	SmartPad Hide	
IDA_SMARTPAD_TOGGLEHIDE	SmartPad Toggle	
IDA_SLEEP_10	Sleep 10msec	
IDA_SLEEP_50	Sleep 50msec	

Symbolic Name	**Friendly Name**	**Description**
IDA_SLEEP_200	Sleep 200msec	
IDA_SLEEP_1000	Sleep 1sec	
IDA_SLEEP_5000	Sleep 5sec	
IDA_SLEEP_20000	Sleep 20sec	
IDA_SLEEP_100000	Sleep 100sec	
IDA_SCAN_RESUME	Scan Resume	Allow scanning
IDA_SCAN_SUSPEND	Scan Suspend	Suspend scanning
IDA_SCAN_TRIGGER	Scan Trigger	Soft trigger scanner
IDA_MACRO_OPEN	Macro Show	Show Macro Tool
IDA_MACRO_CLOSE	Macro Hide	Hide Macro Tool
IDA_MACRO_TOGGLEHIDE	Macro Toggle	Toggle Macro Tool hiding
IDA_MACRO_RECORD	Macro Record	Start Macro record
IDA_MACRO_STOP	Macro Stop	Stop Macro record
IDA_MACRO_PLAY	Macro Play	Replay Macro
IDA_PRINT_SCREEN	Print Screen	Print current screen
IDA_OIA_HIDE	OIA Hide	Hide IBM OIA bar
IDA_OIA_VISIBLE	OIA Show	Show IBM OIA bar
IDA_OIA_TOGGLEHIDE	OIA Toggle	Toggle OIA bar visibility
General IBM and VT Actions		
IDA_PF1	F1	(Not VT PF1)
IDA_PF2	F2	(Not VT PF2)
IDA_PF3	F3	(Not VT PF3)
IDA_PF4	F4	(Not VT PF4)
…	…	
IDA_PF24	F24	
IDA_HOME	Home	
IDA_DOWN	Down	
IDA_UP	Up	
IDA_LEFT	Left	
IDA_RIGHT	Right	
IDA_ENTER	Enter	
IBM Actions		

Symbolic Name	**Friendly Name**	**Description**
IDA_IBM_HOME	IBM Home	
IDA_DELETE	Delete	
IDA_INSERT_ON	Insert On	
IDA_INSERT_OFF	Insert Off	
IDA_INSERT_TOGGLE	Insert Toggle	
IDA_ATTN	Attn	
IDA_CLEAR	Clear	
IDA_CURSOR_SELECT	Cursor Select	
IDA_DUP	DUP	
IDA_ERASE_EOF	Erase EOF	
IDA_ERASE_INPUT	Erase Input	
IDA_FIELD_MARK	Field Mark	
IDA_NEWLINE	Newline	
IDA_PA1	PA1	
IDA_PA2	PA2	
IDA_PA3	PA3	
IDA_RESET	Reset	
IDA_SYSREQ	Sys Request	
5250 Specific Actions		
IDA_FIELD_EXIT	Field Exit	
IDA_FIELD_PLUS	Field +	
IDA_FIELD_MINUS	Field -	
IDA_FIELD_ADVANCE	Field Advance	
IDA_FIELD_BACKSPACE	Field Backspace	
IDA_FIELD_SUB	Field SUB	
IDA_HELP	IBM Help	
IDA_ROLL_DOWN	Roll Down	
IDA_ROLL_UP	Roll Up	
IDA_ROLL_LEFT	Roll Left	
IDA_ROLL_RIGHT	Roll Right	
IDA_BACKSPACE	Backspace	
IDA_PRINT	IBM Print	
VT Actions		
IDA_ANSWERBACK	Answerback	
IDA_FIND	Find	
IDA_INSERT_HERE	Insert Here	
IDA_NEXT	Next	
IDA_PREVIOUS	Previous	

Symbolic Name	**Friendly Name**	**Description**
IDA_REMOVE	Remove	
IDA_SELECT	Select	
IDA_VT_PF1	VT PF1	Numpad PF1 key
IDA_VT_PF2	VT PF2	Numpad PF2 key
IDA_VT_PF3	VT PF3	Numpad PF3 key
IDA_VT_PF4	VT PF4	Numpad PF4 key
IDA_VT_COMMA	Numpad Comma	
IDA_NUMPAD_0	Numpad 0	
IDA_NUMPAD_1	Numpad 1	
IDA_NUMPAD_2	Numpad 2	
IDA_NUMPAD_3	Numpad 3	
IDA_NUMPAD_4	Numpad 4	
IDA_NUMPAD_5	Numpad 5	
IDA_NUMPAD_6	Numpad 6	
IDA_NUMPAD_7	Numpad 7	
IDA_NUMPAD_8	Numpad 8	
IDA_NUMPAD_9	Numpad 9	
IDA_VT_ENTER	Numpad Enter	
IDA_VT_MINUS	Numpad Minus	
IDA_VT_PERIOD	Numpad Period	
IDA_UDK_F6	UDK F6	VT User Defined Key F6
IDA_UDK_F7	UDK F7	VT User Defined Key F7
…	…	
IDA_UDK_F20	UDK F20	VT User Defined Key F20
IDA_VT_HELP	VT Help	
IDA_VT_DO	VT Do	
IDA_ADD	Add	
IDA_MULTIPLY	Multiply	
IDA_DIVIDE	Divide	
Custom VT Sequences		
IDA_VT_SAP0135	VT SAP0135	0x00 0x35
IDA_VT_CSI_M	VT CSI M	ESC [M
IDA_VT_CSI_N	VT CSI N	ESC [N
IDA_VT_CSI_O	VT CSI O	
IDA_VT_CSI_P	VT CSI P	

Symbolic Name	**Friendly Name**	**Description**
IDA_VT_CSI_Q	VT CSI Q	
IDA_VT_CSI_R	VT CSI R	
IDA_VT_CSI_S	VT CSI S	
IDA_VT_CSI_T	VT CSI T	
Windows App Keys		
IDA_APPKEY_K1	App Key 1	
IDA_APPKEY_K2	App Key 2	
…	…	
IDA_APPKEY_K16	App Key 16	
IDA_SCROLL_UPPERLEFT	Scroll Upper Left	
IDA_SCROLL_UPPERRGHT	Scroll Upper Right	
IDA_SCROLL_LOWERLEFT	Scroll Lower Left	
IDA_SCROLL_LOWERRGHT	Scroll Lower Right	
IDA_SCROLL_CENTER	Scroll Center	
IDA_SCROLL_CURSOR_CENTER	Scroll Cursor Center	
IDA_SCROLL_CURSOR_VISIBLE	Scroll Cursor Visible	
IDA_COPYALL	Copy All	Copy screen to clipboard
IDA_PASTE	Paste	Past clipboard
IDA_USTRING_1	Text 1	Send user text 1
IDA_USTRING_2	Text 2	Send user text 2
…	…	…
IDA_USTRING_64	Text 64	Send user text 64
IDA_SCRIPT_1	Script 1	Run Script 1
IDA_SCRIPT_2	Script 2	Run Script 2
…	…	…
IDA_SCRIPT_64	Script 64	Run Script 64
IDA_SIP_HIDE	SIP Hide	
IDA_SIP_SHOW	SIP Show	
IDA_SIP_TOGGLEHIDE	SIP Toggle	
IDA_SIP_LOCKDOWN	SIP Lockdown	
IDA_SIP_UNLOCK	SIP Unlock	
IDA_SIP_UP	SIP Up	
IDA_SIP_DOWN	SIP Down	
IDA_SIP_FORCEDOWN	SIP Forcedown	

Symbolic Name	**Friendly Name**	**Description**
IDA_IM_KEYBOARD	IM Keyboard	
IDA_IM_LOCKED	IM Locked	
HTML Actions		
IDA_URL_HOME	URL Home	
IDA_URL_BACK	URL Back	
IDA_URL	URL	Defines start of URL
Special Actions		
IDA_VIBRATE_100	Vibrate 100ms	
IDA_VIBRATE_200	Vibrate 200ms	
IDA_VIBRATE_500	Vibrate 500ms	
IDA_VIBRATE_1000	Vibrate 1sec	
IDA_VIBRATE_2000	Vibrate 2sec	
IDA_VIBRATE_5000	Vibrate 5sec	
IDA_BEEP_OK	Beep	
IDA_BEEP_WARN	Beep Warn	
IDA_BEEP_LOUD	Beep Loud	
IDA_KBD_ALPHA	KeyMode Alpha	
IDA_KBD_NUMERIC	KeyMode Numeric	
IDA_KBD_ALPHANUM	KeyMode AlphaNum	
IDA_KBD_UPPERALPHA	KeyMode Upper Alpha	
IDA_KBD_LOWERALPHA	Keymode Lower Alpha	
IDA_KBD_FUNCMODE	KeyMode Func	
IDA_KBD_CYCLEMODE	KeyMode Cycle	Cycle to next mode
IDA_POPUP_IPADDRESS	Show IP Address	
IDA_POPUP_MACADDRESS	Show MAC Address	
IDA_POPUP_BATTERY	Show Battery	
IDA_POPUP_TIME	Show Time	
IDA_POPUP_SERIALNUMBER	Show Serial #	
IDA_POPUP_DEVICEID	Show Device ID	
IDA_POPUP_RFINFO	Show RF info	

Appendix 2 - Properties

The properties listed in this appendix may be accessed via the GetProperty and SetProperty methods on the CETerm object. Properties marked (RO) are read-only and may not be set with SetProperty. The symbol T/F indicates a true or false value.

APPLICATION PROPERTIES

Property Name	Description
app.buildid (RO)	Program build identifier
app.commandline (RO)	Commandline starting program
app.name (RO)	Program name
app.script.NN	Script # NN contents, NN is 1-64
app.session.active (RO)	Currently active session
app.usertext.NN	User text # NN contents, NN is 1-64
app.version (RO)	Program version

DEVICE PROPERTIES

Property Name	Description
device.batterystatus (RO) device.battery.statustext (RO)	Current battery status string
device.battery.status (RO)	Current battery status -1 – unknown, 0 – critical, 1 – warning, 2 – low, 3 – medium, 4 – high, 5 - charging
device.battery.level (RO)	Current battery strength - 0 – 100 -1 – unknown
device.bluetoothaddress (RO)	Bluetooth address
device.deviceid (RO)	Device ID string
device.ipaddress (RO)	IP Address of handheld
device.macaddress (RO)	MAC Address of handheld
device.oeminfo (RO)	OEM information text
device.platformid (RO)	Windows CE Platform ID

Property Name	Description
device.presetid (RO)	Windows CE Preset ID
device.rf.strength (RO)	RF signal strength 0-100, -2 – not associated with AP, -1 – unknown
device.rf.status (RO)	RF status -1 – unknown, 0 – unassociated, 1 – poor, 2 – fair, 3 – good, 4 – very good, 5 – excellent
device.serialnumber (RO)	Device serialnumber

SESSION PROPERTIES

Session properties begin with “sessionX” where X is 1 through 4. For example “session4.connection.host”. If no ‘X’ value is found, the currently active session number is used.

Property Name	Description
sessionX.connection.host	Session host (or home URL)
sessionX.connection.port	TE session port
sessionX.connection.type	Session type 3270, 5250, VT220, HTML
sessionX.printer.network.queue	Network printing queue
sessionX.printer.serial.port	Serial printing port

Scanner Properties

Scanner properties are unique for each session. Scanner properties begin with "sessionX.scanner" where X is 1 through 4. For example "session4.scanner.enabled". If no 'X' value is found, the currently active session number is used. We use the name "scanner" for all types of barcode readers, including laser scanners and imagers. If a hardware vendor is listed, the property is specific to barcode readers made by that vendor.

NOTE: If you are changing the scanner properties for the currently activesession, you must call CETerm.PostIDA("IDA_SCAN_APPLYCONFIG", 0); for the changes to take effect.

WARNING: Not all properties are applicable to all hardware devices. Different devices may use different names to refer to the same parameters. You should look at the settings available in the CETerm configuration dialogs to determine if the property is appropriate and what values it may accept.

Property Name	**Description**
sessionX.scanner.enabled	Scanner is enabled. T/F
sessionX.scanner.aimerenabled	Aimer is enabled. T/F
sessionX.scanner.wedgeenabled	Allow wedge if scanner disabled in CETerm (Intermec). T/F
sessionX.scanner.focusnear	Imager focus near if true. T/F
sessionX.scanner.enhanced1d	Improved decode for poor quality barcodes
sessionX.scanner.negativelabel	Improved decode negative barcodes T/F
sessionX.scanner.picklistmode	Decode barcode under cross-hairs. T/F
sessionX.scanner.preamble	Barcode preamble
sessionX.scanner.postamble	Barcode postamble
sessionX.scanner.grid	Barcode grid filter (Intermec) Use OnBarcodeRead for more features.
sessionX.scanner.beamtimeout	Scan beam timeout, milliseconds
sessionX.scanner.aimertimeout	Aimer timeout, milliseconds
sessionX.scanner.aimmode	Aim mode. none, dot, slab, reticle
sessionX.scanner.redundancy	Linear security/redundancy, 0-5

Common Symbology Properties

Symbology properties are unique for each session. Symbology properties begin with “sessionX.scanner.SSS” where X is 1 through 4 and SSS represents a symbology name and may be 3 or more characters long. For example “session4.scanner.upca.enabled”. If no ‘X’ value is found, the currently active session number is used. See the Symbology Names table below for SSS values.

NOTE: If you are changing the scanner properties for the currently activesession, you must call CETerm.PostIDA(“IDA_SCAN_APPLYCONFIG”, 0); for the changes to take effect.

WARNING: Not all properties are applicable to all hardware devices or all symbologies. Different devices may use different names to refer to the same parameters. You should look at the settings available in the CETerm configuration dialogs to determine if the property is appropriate and what values it may accept. This is also true for the symbologies that a device supports. You may be able to successfully change a parameter that is not supported on a device.

Last Property Level	**Description**
enabled	Symbology is enabled. T/F
verifycheck	Require check digit validation. T/F
redundancy	Scan redundancy flag. T/F (Symbol)
reportcheck	Report the check digit with the data. T/F
reportnumbersystem	Report UPC number system. T/F
reportcountry	Report UPC country code. T/F
reportstartstop	Report start/stop digits with barcode data. T/F
converttoupca	Convert barcode output to UPCA. T/F
converttoean13	Convert barcode output to EAN-13. T/F
supplemental2	Enable 2 digit supplemental or add-on barcode. T/F
supplemental5	Enable 5 digit supplemental or add-on barcode. T/F
supplementalrequired	Require supplemental on UPC. T/F
supplementalseparator	Insert supplemental separator. T/F
addendum	Supplemental mode. none, optional, required
minlength	Minimum barcode length. Not supported by all symbologies. See configuration dialogs for ranges.
maxlength	Maximum barcode length. Not supported by all symbologies. See configuration dialogs for ranges.
stripleading	Strip characters from start of barcode. 0-32
striptrailing	Strip characters from end of barcode. 0-32
customid	Custom symbology ID. 4 character string

CODABAR SYMBOLOGY PROPERTIES

Codabar specific symbology properties are unique for each session. Symbology properties begin with "sessionX.scanner.codabar" where X is 1 through. For example "session4.scanner.codabar.clsiediting". If no 'X' value is found, the currently active session number is used.

NOTE: If you are changing the symbology properties for the currently active session, you must call "CETerm.PostIDA("IDA_SCAN_APPLYCONFIG", 0) for the changes to take effect.

WARNING: Not all properties are applicable to all hardware devices. Different devices may use different names to refer to the same parameters. You should look at the settings available in the CETerm configuration dialogs to determine if the property is appropriate and what values it may accept.

Last Property Level	**Description**
clsiediting	CLSI editing is enabled. T/F
notisediting	NOTIS editing is enabled. T/F
startstop	Start/Stop digit modes. Not all modes apply to all devices. See CETerm configuration for values on a specific device. discard, none, abcd, dc1-dc4, lowerabcd, abcd/tn*e, aa, bb, cc, dd, any

CODE39 SYMBOLOGY PROPERTIES

Code 39 specific symbology properties are unique for each session. Symbology properties begin with "sessionX.scanner.code39" where X is 1 through. For example "session4.scanner.code39.clsiediting". If no 'X' value is found, the currently active session number is used.

NOTE: If you are changing the symbology properties for the currently active session, you must call "CETerm.PostIDA("IDA_SCAN_APPLYCONFIG", 0) for the changes to take effect.

WARNING: Not all properties are applicable to all hardware devices. Different devices may use different names to refer to the same parameters. You should look at the settings available in the CETerm configuration dialogs to determine if the property is appropriate and what values it may accept.

Last Property Level	**Description**
asciimode	Select ASCII mode. Not all modes apply to all devices. See CETerm configuration for values on a specific device. base, full, mixedfull
fullascii	Enable Full-ASCII mode. T/F
verifycheck39	Check digit validation mode. 0-255
reportstartstop	Report start/stop with barcode. T/F
convertocode32	Convert to Code 32 format. T/F
reportcode32prefix	Report Code 32 prefix with barcode. T/F
concatenation	Enable concatenation. T/F
stripAIAG	Remove AIAG codes. T/F
erroraccept	Allow format error. T/F

Code 128 Symbology Properties

Code 128 specific symbology properties are unique for each session. Symbology properties begin with “sessionX.scanner.code128” where X is 1 through. For example “session4.scanner.code128.ISBT”. If no ‘X’ value is found, the currently active session number is used.

NOTE: If you are changing the symbology properties for the currently active session, you must call “CETerm.PostIDA(“IDA_SCAN_APPLYCONFIG”, 0) for the changes to take effect.

WARNING: Not all properties are applicable to all hardware devices. Different devices may use different names to refer to the same parameters. You should look at the settings available in the CETerm configuration dialogs to determine if the property is appropriate and what values it may accept.

Last Property Level	**Description**
FNC1char	FNC1 character. 0-255
CIP	Enable CIP labels. T/F
ISBT	Enable ISBT 128 labels. T/F
other	Enable other 128 labels. T/F
UCCEAN	Enable UCCEAN 128 labels. T/F

UPC-EAN General Symbology Properties

UPC-EAN general symbology properties are unique for each session. Symbology properties begin with “sessionX.scanner.upc-ean” where X is 1 through. For example “session4.scanner.upc-ean.bookland”. If no ‘X’ value is found, the currently active session number is used.

NOTE: If you are changing the symbology properties for the currently active session, you must call “CETerm.PostIDA(“IDA_SCAN_APPLYCONFIG”, 0) for the changes to take effect.

WARNING: Not all properties are applicable to all hardware devices. Different devices may use different names to refer to the same parameters. You should look at the settings available in the CETerm configuration dialogs to determine if the property is appropriate and what values it may accept.

Last Property Level	Description
bookland	Enable Bookland labels. T/F
coupon	Enable Coupon labels. T/F
lineardecode	Enable linear decode. T/F
supplemental2	Enable 2 digit supplemental or add-on barcode. T/F
supplemental5	Enable 5 digit supplemental or add-on barcode. T/F
supplementalretry	Supplemental decode retry count. 2-10
randomweightcheckdigit	Enable random weight check digit. T/F
supplementalmode	Supplemental mode. none, always, auto
securitylevel	Decode security level. none, all, ambiguous

Symbology Names

Symbology properties begin with “sessionX.scanner.SSS” where X is 1 through 4 and SSS represents a symbology name and may be 3 or more characters long. The following table lists all available symbology names.

WARNING: Not all symbologies are applicable to all hardware. Different devices may use different names to refer to similar symbologies, e.g., upce and upce0. You should look at the symbologies available in the CETerm configuration dialogs to determine the correct name.

Symbology Name	Description
ames	Ames
auspostal	Australian Postal
aztec	Aztec
bpo	British Postal
canpostal	Canadian Postal
chinapostal	China Postal
codabar	Codabar
codablock	Codablock
code11	Code 11
code16k	Code 16k
code32	Code 32
code39	Code 39
code49	Code 49
code93	Code 93
code128	Code 128
composite	Composite AB and C
couponcode	Coupon Code
d2of5	Discrete (standard) 2 of 5
datamatrix	Datamatrix
delta	Delta Code
dutchpostal	Dutch Postal
ean8	EAN-8
ean13	EAN-13
i2of5	Interleaved 2 of 5
iata25	IATA 2 of 5
idtag	ID Tag
isbt	ISBT
japanpostal	Japan Postal
koreapostal	Korea Postal
label45	Label 45
m2of5	Matrix 2 of 5

Symbology Name	**Description**
maxicode	Maxicode
mesa	Mesa
micropdf	Micro PDF
msi	MSI
pdf	PDF 417
pdf417	PDF 417
pharma39	Pharma 39
planet	Planet
plessey	Plessey
posicode	Posicode
postnet	Postnet
qrcode	QR Code
rss	RSS 14
rss14	RSS 14
rssexp	RSS Expanded
rsslim	RSS Limited
rssltd	RSS Limited
telepen	Telepen
tlc39	TLC 39
trioptic39	Trioptic 39
ukpostal	British (UK) Postal
upca	UPC-A
upce	UPC-E
upce0	UPC-E0
upce1	UPC-E1
upc-ean	UPC-EAN General Settings
usplanet	US Planet
uspostnet	US Postnet
usps4cb	USPS 4CB

Appendix 3 – Symbology LabelTypes

This appendix contains a list of symbology labeltypes that are returned in the "type" argument of OnBarcodeRead. These are also available to a ScannerNavigate META tag handler. Please note that not all hardware devices return these values. You may need to test scan a known barcode to find the labeltype value for that barcode.

LabelType Character	**Hexadecimal Value**	**Symbology**
#	0x23	Plessey
&	0x24	Telepen
%	0x25	Codablock A
$	0x26	Codablock F
' (single quote)	0x27	Matrix 2 of 5
(	0x28	Code 49
)	0x29	Code 16K
*	0x2A	Ankercode
+	0x2B	Aztec
, (comma)	0x2C	Korea Postal
0	0x30	UPC-E or UPC-E0
1	0x31	UPC-E1
2	0x32	UPC-A
3	0x33	MSI
4	0x34	EAN-8
5	0x35	EAN-13
6	0x36	Codabar
7	0x37	Code 39
8	0x38	Discrete 2 of 5
9	0x39	Interleaved 2 of 5
: (colon)	0x3A	Code 11
; (semi-colon)	0x3B	Code 93
<	0x3C	Code 128
>	0x3E	IATA 2 of 5
?	0x3F	EAN 128

LabelType Character	Hexadecimal Value	Symbology
@	0x40	PDF 417
A	0x41	ISBT 128
B	0x42	Trioptic 39
C	0x43	Coupon Code
D	0x44	Bookland
E	0x45	Micro PDF
F	0x46	Code 32
G	0x47	Macro PDF
H	0x48	Maxicode
I	0x49	Datamatrix
J	0x4A	QR Code
K	0x4B	Macro Micro PDF
L	0x4C	RSS-14
M	0x4D	RSS Limited
N	0x4E	RSS Expanded
V	0x56	Composite AB
W	0x57	Composite C
X	0x58	TLC 39
a	0x61	US Postnet
b	0x62	US Planet
c	0x63	UK (British) Postal
d	0x64	Japan Postal
e	0x65	Australian Postal
f	0x66	Dutch Postal
g	0x67	Canadian Postal
p	0x70	Mesa
q	0x71	OCR
r	0x72	China Postal
s	0x73	Posicode
t	0x74	USPS4CB
u	0x75	ID Tag

Appendix 4 - Constants

This appendix contains various constants that are used by CETerm Automation Objects. Many of these constants are a direct representation of the equivalent values from the Windows CE system APIs and constants.

These constants are mostly presented as global JavaScript variables for direct inclusion in scripts. For simple scripts, a few extra global variables are not harmful, but good programming practices seek to minimize globals to prevent accidental name collisions. Some constants below are represented more appropriately as properties of "constant" objects.

For efficiency, you should not include constant definitions that are not used by your scripts. One good pattern is to re-express the needed constants as members of a single CONSTANTS object. Here is a brief example showing how a single CONTANTS object might contain multiple categories of constants.

```
// Express constants as object properties.
var CONSTANTS = {
  MESSAGEBOX: {FLAG_OK:0x00000000, FLAG_OKCANCEL:0x00000001},
  FILE_ATTRIBUTE: {READONLY:0x00000001, HIDDEN:0x00000002},
  SERIAL_PORT: {NOPARITY:0x00, ODDPARITY:0x01, EVENPARITY:0x02}
}

// Refer to constants using normal JavaScript syntax
var myparity = CONSTANTS.SERIAL_PORT.ODDPARITY;
```

BATTERY AND POWER MANAGEMENT CONSTANTS

```
// Power Management and Battery Constants
//
var AC_LINE_OFFLINE            = 0x00;
var AC_LINE_ONLINE             = 0x01;
var AC_LINE_BACKUP_POWER       = 0x02;
var AC_LINE_UNKNOWN            = 0xFF;

var BATTERY_FLAG_HIGH          = 0x01;
var BATTERY_FLAG_LOW           = 0x02;
var BATTERY_FLAG_CRITICAL      = 0x04;
var BATTERY_FLAG_CHARGING      = 0x08;
var BATTERY_FLAG_NO_BATTERY    = 0x80;
var BATTERY_FLAG_UNKNOWN       = 0xFF;

var BATTERY_PERCENTAGE_UNKNOWN = 0xFF;

var BATTERY_LIFE_UNKNOWN       = 0xFFFFFFFF;

var BATTERY_CHEMISTRY_ALKALINE = 0x01;
var BATTERY_CHEMISTRY_NICD     = 0x02;
var BATTERY_CHEMISTRY_NIMH     = 0x03;
var BATTERY_CHEMISTRY_LION     = 0x04;
var BATTERY_CHEMISTRY_LIPOLY   = 0x05;
var BATTERY_CHEMISTRY_ZINCAIR  = 0x06;
var BATTERY_CHEMISTRY_UNKNOWN  = 0xFF;

// Power State
var POWER_STATE_UNKNOWN      = -1; // Unknown
var POWER_STATE_FULL_ON      = 0;  // Full power
var POWER_STATE_LOW_POWER_ON = 1;  // Functional at low power
var POWER_STATE_STANDBY      = 2;  // Partial power, auto-wake
var POWER_STATE_SLEEP        = 3;  // Partial power, manual-wake
var POWER_STATE_OFF          = 4;  // Unpowered
```

BROWSER ERROR CONSTANTS

```
// Navigate Error HRESULT status codes
// See Microsoft SDK for documentation.
//
// URL string is not valid.
var INET_E_INVALID_URL               = 0x800C0002;
// No session found.
var INET_E_NO_SESSION                = 0x800C0003;
// Unable to connect to server.
var INET_E_CANNOT_CONNECT            = 0x800C0004;
// Requested resource is not found.
var INET_E_RESOURCE_NOT_FOUND        = 0x800C0005;
// Requested object is not found.
var INET_E_OBJECT_NOT_FOUND          = 0x800C0006;
// Requested data is not available.
var INET_E_DATA_NOT_AVAILABLE        = 0x800C0007;
// Failure occurred during download.
var INET_E_DOWNLOAD_FAILURE          = 0x800C0008;
// Authentication required.
var INET_E_AUTHENTICATION_REQUIRED   = 0x800C0009;
// Required media not available or valid.
var INET_E_NO_VALID_MEDIA            = 0x800C000A;
// Connection timed out.
var INET_E_CONNECTION_TIMEOUT        = 0x800C000B;
// Request is invalid.
var INET_E_INVALID_REQUEST           = 0x800C000C;
// Protocol is not recognized.
var INET_E_UNKNOWN_PROTOCOL          = 0x800C000D;
// Failed due to security issue.
var INET_E_SECURITY_PROBLEM          = 0x800C000E;
// Unable to load data from the server.
var INET_E_CANNOT_LOAD_DATA          = 0x800C000F;
// Unable to create an instance of the object.
var INET_E_CANNOT_INSTANTIATE_OBJECT = 0x800C0010;
```

```
    // Attempt to redirect the navigation failed.
    var INET_E_REDIRECT_FAILED           = 0x800C0014;
    // Navigation redirected to a directory.
    var INET_E_REDIRECT_TO_DIR           = 0x800C0015;
    // Unable to lock request with the server.
    var INET_E_CANNOT_LOCK_REQUEST       = 0x800C0016;
    // Reissue request with extended binding.
    var INET_E_USE_EXTEND_BINDING        = 0x800C0017;
    // Binding is terminated.
    var INET_E_TERMINATED_BIND           = 0x800C0018;
    // Permission to download is declined.
    var INET_E_CODE_DOWNLOAD_DECLINED    = 0x800C0100;
    // Result is dispatched.
    var INET_E_RESULT_DISPATCHED         = 0x800C0200;
    // Cannot replace a protected SFP file.
    var INET_E_CANNOT_REPLACE_SFP_FILE   = 0x800C0300;
```

FILE ATTRIBUTE CONSTANTS

```
    // File attribute flags
    // See Microsoft SDK for documentation.
    var FILE_ATTRIBUTE_READONLY     = 0x00000001;
    var FILE_ATTRIBUTE_HIDDEN       = 0x00000002;
    var FILE_ATTRIBUTE_SYSTEM       = 0x00000004;
    var FILE_ATTRIBUTE_DIRECTORY    = 0x00000010;
    var FILE_ATTRIBUTE_ARCHIVE      = 0x00000020;
    var FILE_ATTRIBUTE_INROM        = 0x00000040;
    var FILE_ATTRIBUTE_ENCRYPTED    = 0x00000040;
    var FILE_ATTRIBUTE_NORMAL       = 0x00000080;
    var FILE_ATTRIBUTE_TEMPORARY    = 0x00000100;
    var FILE_ATTRIBUTE_COMPRESSED   = 0x00000800;
    var FILE_ATTRIBUTE_ROMSTATICREF = 0x00001000;
    var FILE_ATTRIBUTE_ROMMODULE    = 0x00002000;
```

IBM STATUS CONSTANTS

```
    // Constants used by DisplayStatus
    var IBM_STATUS_UNKNOWN      = 0;
    var IBM_STATUS_SENDING      = 1;
    var IBM_STATUS_WAITING      = 2;
    var IBM_STATUS_SYSTEM       = 3;
    var IBM_STATUS_PROTECTED    = 4;
    var IBM_STATUS_NUMERIC      = 5;
    var IBM_STATUS_FULL         = 6;
    var IBM_STATUS_INSERT       = 7;
    var IBM_STATUS_SYSCLEAR     = 8;
    var IBM_STATUS_WAITCLEAR    = 9;
```

KEYBOARD CONSTANTS

```
// Constants used by OnKeyboardStateChange
var IBM_KEYBOARD_HARDWARE_ERROR  = 0;
var IBM_KEYBOARD_NORMAL_LOCKED   = 1;
var IBM_KEYBOARD_NORMAL_UNLOCKED = 2;
var IBM_KEYBOARD_POWER_ON        = 3;
var IBM_KEYBOARD_PRE_HELP_ERROR  = 4;
var IBM_KEYBOARD_POST_HELP_ERROR = 5;
var IBM_KEYBOARD_SS_MESSAGE      = 6;
var IBM_KEYBOARD_SYSTEM_REQUEST  = 7;

var VT_KEYBOARD_LOCKED   = 1;
var VT_KEYBOARD_UNLOCKED = 2;


// Keyboard Hotkey Constants
var HOTKEY_MODIFIERS =
{
MOD_ALT     : 0x1,      // Either ALT key must be held down.
MOD_CONTROL : 0x2,      // Either CTRL key must be held down.
MOD_SHIFT   : 0x4,      // Either SHIFT key must be held down.
MOD_WIN     : 0x8,      // Either WINDOWS key was held down.
MOD_KEYUP   : 0x1000    // Both key up events and key down events
                        // generate a WM_HOTKEY message.
};


// Key state flags
var KEY_STATE_FLAGS =
{
KeyStateToggledFlag       : 0x0001,     //  Key is toggled.
KeyStateGetAsyncDownFlag  : 0x0002,     //  Key went down since last
                                        //  GetAsyncKey call.
KeyStatePrevDownFlag      : 0x0040,     //  Key was previously down.
KeyStateDownFlag          : 0x0080,     //  Key is currently down.

KeyShiftAnyCtrlFlag       : 0x40000000, //  L or R control is down.
KeyShiftAnyShiftFlag      : 0x20000000, //  L or R shift is down.
KeyShiftAnyAltFlag        : 0x10000000, //  L or R alt is down.
KeyShiftCapitalFlag       : 0x08000000, //  VK_CAPITAL is toggled.
KeyShiftLeftCtrlFlag      : 0x04000000, //  L control is down.
KeyShiftLeftShiftFlag     : 0x02000000, //  L shift is down.
KeyShiftLeftAltFlag       : 0x01000000, //  L alt is down.
KeyShiftLeftWinFlag       : 0x00800000, //  L Win key is down.
KeyShiftRightCtrlFlag     : 0x00400000, //  R control is down.
KeyShiftRightShiftFlag    : 0x00200000, //  R shift is down.
KeyShiftRightAltFlag      : 0x00100000, //  R alt is down.
KeyShiftRightWinFlag      : 0x00080000, //  R Win key is down.
KeyShiftDeadFlag          : 0x00020000, //  Char is dead char.
KeyShiftNoCharacterFlag   : 0x00010000, //  No corresponding char.

KeyShiftNumLockFlag       : 0x00001000, //  NumLock toggled state.
KeyShiftScrollLockFlag    : 0x00000800  //  ScrollLock toggled state.
};
```

MESSAGEBOX CONSTANTS

```
// MessageBox flags
// See Microsoft SDK for documentation.
var MESSAGEBOX_FLAG_OK               = 0x00000000;
var MESSAGEBOX_FLAG_OKCANCEL         = 0x00000001;
var MESSAGEBOX_FLAG_ABORTRETRYIGNORE = 0x00000002;
var MESSAGEBOX_FLAG_YESNOCANCEL      = 0x00000003;
var MESSAGEBOX_FLAG_YESNO            = 0x00000004;
var MESSAGEBOX_FLAG_RETRYCANCEL      = 0x00000005;

var MESSAGEBOX_FLAG_ICONERROR       = 0x00000010;
var MESSAGEBOX_FLAG_ICONQUESTION    = 0x00000020;
var MESSAGEBOX_FLAG_ICONWARNING     = 0x00000030;
var MESSAGEBOX_FLAG_ICONINFORMATION = 0x00000040;

var MESSAGEBOX_FLAG_DEFBUTTON1 = 0x00000000;
var MESSAGEBOX_FLAG_DEFBUTTON2 = 0x00000100;
var MESSAGEBOX_FLAG_DEFBUTTON3 = 0x00000200;
var MESSAGEBOX_FLAG_DEFBUTTON4 = 0x00000300;

var MESSAGEBOX_FLAG_APPLMODAL     = 0x00000000;
var MESSAGEBOX_FLAG_SETFOREGROUND = 0x00010000;
var MESSAGEBOX_FLAG_TOPMOST       = 0x00040000;

// MessageBox returned values
var MESSAGEBOX_IDOK     = 1;
var MESSAGEBOX_IDCANCEL = 2;
var MESSAGEBOX_IDABORT  = 3;
var MESSAGEBOX_IDRETRY  = 4;
var MESSAGEBOX_IDIGNORE = 5;
var MESSAGEBOX_IDYES    = 6;
var MESSAGEBOX_IDNO     = 7;
```

PLAYSOUND CONSTANTS

```
// PlaySound flags
// See Microsoft SDK for documentation.
var PLAYSOUND_FLAG_ASYNC = 0x00000001;  // Play asynchronously
var PLAYSOUND_FLAG_NODEFAULT = 0x00000002;  // No default sound
var PLAYSOUND_FLAG_LOOP = 0x00000008;  // Repeat play, needs ASYNC.
var PLAYSOUND_FLAG_NOSTOP = 0x00000010;  // Don’t stop current sound
var PLAYSOUND_FLAG_NOWAIT = 0x00002000;  // Don’t play if driver busy
```

REGISTRY CONSTANTS

```
// Registry constants
// See Microsoft SDK for documentation.
// Root key names
var HKEY_CLASSES_ROOT  = "HKEY_CLASSES_ROOT";
var HKEY_CURRENT_USER  = "HKEY_CURRENT_USER";
var HKEY_LOCAL_MACHINE = "HKEY_LOCAL_MACHINE";
var HKEY_USERS         = "HKEY_USERS";

// Data types
var REG_SZ             = "REG_SZ";
var REG_DWORD          = "REG_DWORD";
var REG_BINARY         = "REG_BINARY";
var REG_MULTI_SZ       = "REG_MULTI_SZ";
var REG_EXPAND_SZ      = "REG_EXPAND_SZ";

// Returned Status
var REGISTRY_SUCCESS         = 0;
var REGISTRY_FAIL            = -1;
var REGISTRY_BAD_HIVE        = -2;
var REGISTRY_BAD_KEYNAME     = -3;
var REGISTRY_BAD_DATATYPE    = -4;
var REGISTRY_BAD_VALUE       = -5;
var REGISTRY_BAD_VALUEFORMAT = -6;
var REGISTRY_OUTOFMEMORY     = -7;
```

SERVICE STATE CONSTANTS

```
// Service state for GPS and other devices
// See msdn.microsoft.com IOCTL_SERVICE_STATUS for documentation.
var SERVICE_STATE_OFF           = 0; // Service is turned off.
var SERVICE_STATE_ON            = 1; // Service is turned on.
var SERVICE_STATE_STARTING_UP   = 2; // Service is starting up.
var SERVICE_STATE_SHUTTING_DOWN = 3; // Service is shutting down.
var SERVICE_STATE_UNLOADING     = 4; // Service is unloading.
var SERVICE_STATE_UNINITIALIZED = 5; // Service is not uninitialized.
var SERVICE_STATE_UNKNOWN       = 0xffffffff;
```

SERIAL PORT CONSTANTS

```
    // Constants for SerialPort control object

    //
    // DTR Control Flow Values.
    //
    var DTR_CONTROL_DISABLE     = 0x00;
    var DTR_CONTROL_ENABLE      = 0x01;
    var DTR_CONTROL_HANDSHAKE   = 0x02;

    //
    // RTS Control Flow Values
    //
    var RTS_CONTROL_DISABLE     = 0x00;
    var RTS_CONTROL_ENABLE      = 0x01;
    var RTS_CONTROL_HANDSHAKE   = 0x02;
    var RTS_CONTROL_TOGGLE      = 0x03;

    //
    // Parity Modes
    //
    var NOPARITY            = 0x00;
    var ODDPARITY           = 0x01;
    var EVENPARITY          = 0x02;
    var MARKPARITY          = 0x03;
    var SPACEPARITY         = 0x04;

    //
    // Stop Bit Counts
    //
    var ONESTOPBIT          = 0x00;
    var ONE5STOPBITS        = 0x01;
    var TWOSTOPBITS         = 0x02;

    //
    // Baud rates
    //
    var CBR_110             = 110;
    var CBR_300             = 300;
    var CBR_600             = 600;
    var CBR_1200            = 1200;
    var CBR_2400            = 2400;
    var CBR_4800            = 4800;
    var CBR_9600            = 9600;
    var CBR_14400           = 14400;
    var CBR_19200           = 19200;
    var CBR_38400           = 38400;
    var CBR_56000           = 56000;
    var CBR_57600           = 57600;
    var CBR_115200          = 115200;
```

```
var CBR_128000          = 128000;
var CBR_256000          = 256000;

//
// Error Flags
//
var CE_RXOVER           = 0x0001;   // Receive Queue overflow
var CE_OVERRUN          = 0x0002;   // Receive Overrun Error
var CE_RXPARITY         = 0x0004;   // Receive Parity Error
var CE_FRAME            = 0x0008;   // Receive Framing error
var CE_BREAK            = 0x0010;   // Break Detected
var CE_TXFULL           = 0x0100;   // TX Queue is full
var CE_PTO              = 0x0200;   // LPTx Timeout
var CE_IOE              = 0x0400;   // LPTx I/O Error
var CE_DNS              = 0x0800;   // LPTx Device not selected
var CE_OOP              = 0x1000;   // LPTx Out-Of-Paper
var CE_MODE             = 0x8000;   // Requested mode unsupported

//
// Access
//
var GENERIC_READ        = 0x80000000;
var GENERIC_WRITE       = 0x40000000;

//
// Events
//
var EV_RXCHAR           = 0x0001;   // Any Character received
var EV_RXFLAG           = 0x0002;   // Received certain character
var EV_TXEMPTY          = 0x0004;   // Transmitt Queue Empty
var EV_CTS              = 0x0008;   // CTS changed state
var EV_DSR              = 0x0010;   // DSR changed state
var EV_RLSD             = 0x0020;   // RLSD changed state
var EV_BREAK            = 0x0040;   // BREAK received
var EV_ERR              = 0x0080;   // Line status error occurred
var EV_RING             = 0x0100;   // Ring signal detected
var EV_PERR             = 0x0200;   // Printer error occured
var EV_RX80FULL         = 0x0400;   // Receive buffer is 80% full
var EV_EVENT1           = 0x0800;   // Provider specific event 1
var EV_EVENT2           = 0x1000;   // Provider specific event 2
var EV_POWER            = 0x2000;   // WINCE Power event.

var EVENT_WAIT_FAILED   = 0x01000000;   // Wait failed, see LastError
var EVENT_WAIT_CANCELED = 0x02000000;   // Canceled by user

//
// Extended Functions codes
//
var SETXOFF             = 1;        // Simulate XOFF received
var SETXON              = 2;        // Simulate XON received
var SETRTS              = 3;        // Set RTS high
var CLRRTS              = 4;        // Set RTS low
var SETDTR              = 5;        // Set DTR high
var CLRDTR              = 6;        // Set DTR low
// Gap for NT code RESETDEV, not supported on CE
var SETBREAK            = 8;        // Set the device break line.
var CLRBREAK            = 9;        // Clear the device break line.

// Some devices share a UART between an IRDA port and a serial port.
// These escape functions allow control over the mode.
var SETIR               = 10;       // Set the port to IR mode.
var CLRIR               = 11;       // Set the port to non-IR mode.

//
// Purge mode flags.
//
var PURGE_TXCLEAR = 0x0004;   // Kill the transmit queue.
var PURGE_RXCLEAR = 0x0008;   // Kill the receive queue.

//
// Modem Status Flags
//
var MS_CTS_ON           = 0x0010;
var MS_DSR_ON           = 0x0020;
var MS_RING_ON          = 0x0040;
var MS_RLSD_ON          = 0x0080;
```

WINDOW CONSTANTS

```
// Special window handle for broadcast
var HWND_BROADCAST = 0xFFFFFFFF;

var WINDOW_RELATIONS =
{
HWNDFIRST : 0x0,
 // The window of the same type that is highest in the z-order.
 // If the specified window is a topmost window, the handle
 // identifies the topmost window that is highest in the z-order.
 // If the specified window is a child window, the handle
 // identifies the sibling window that is highest in the z-order.

HWNDLAST  : 0x1,
 // The window of the same type that is lowest in the z-order.
 // If the specified window is a topmost window, the handle
 // identifies the topmost window that is lowest in the z-order.
 // If the specified window is a child window, the handle
 // identifies the sibling window that is lowest in the z-order.

HWNDNEXT  : 0x2,
 // The window below the specified window in the z-order.
 // If the specified window is a topmost window, the handle
 // identifies the topmost window below the specified window.
 // If the specified window is a child window, the handle
 // identifies the sibling window below the specified window.

HWNDPREV  : 0x3,
 // The window above the specified window in the z-order.
 // If the specified window is a topmost window, the handle
 // identifies the topmost window above the specified window.
 // If the specified window is a child window, the handle
 // identifies the sibling window above the specified window.

OWNER     : 0x4,
 // The specified window's owner window, if any.
 // This flag will not retrieve a parent window.

CHILD     : 0x5
 // The child window at the top of the z-order if the specified
 // window is a parent window; otherwise, the retrieved handle
 // is NULL.
};
```

Appendix 5 – Microsoft Virtual-Key (VK) Codes

This appendix lists the standard Microsoft Virtual-Key codes. These codes are sent to applications when keys are pressed. Most devices generate only a small subset of these codes, depending on the hardware keyboard. Some VK codes are not applicable to Windows CE but are listed for completeness. Many hardware vendors use unassigned values, between 0x01 and 0xFF, for custom behaviors. For additional details, search msdn.microsoft.com with the keywords "virtual key codes" and see your hardware documentation.

Symbolic Name	**Value**
VK_LBUTTON	0x01
VK_RBUTTON	0x02
VK_CANCEL	0x03
VK_MBUTTON	0x04
VK_XBUTTON1	0x05
VK_XBUTTON2	0x06
VK_BACK	0x08
VK_TAB	0x09
VK_CLEAR	0x0C
VK_RETURN	0x0D
VK_SHIFT	0x10
VK_CONTROL	0x11
VK_MENU	0x12
VK_PAUSE	0x13
VK_CAPITAL	0x14
VK_KANA	0x15
VK_JUNJA	0x17
VK_FINAL	0x18
VK_HANJA	0x19
VK_ESCAPE	0x1B
VK_CONVERT	0x1C
VK_NONCONVERT	0x1D
VK_ACCEPT	0x1E
VK_MODECHANGE	0x1F
VK_SPACE	0x20
VK_PRIOR	0x21
VK_NEXT	0x22
VK_END	0x23
VK_HOME	0x24
VK_LEFT	0x25

Symbolic Name	Value
VK_UP	0x26
VK_RIGHT	0x27
VK_DOWN	0x28
VK_SELECT	0x29
VK_PRINT	0x2A
VK_EXECUTE	0x2B
VK_SNAPSHOT	0x2C
VK_INSERT	0x2D
VK_DELETE	0x2E
VK_HELP	0x2F
0	0x30
1	0x31
2	0x32
3	0x33
4	0x34
5	0x35
6	0x36
7	0x37
8	0x38
9	0x39
A	0x41
B	0x42
C	0x43
D	0x44
E	0x45
F	0x46
G	0x47
H	0x48
I	0x49
J	0x4A
K	0x4B
L	0x4C
M	0x4D
N	0x4E
O	0x4F
P	0x50
Q	0x51
R	0x52
S	0x53
T	0x54
U	0x55

Symbolic Name	**Value**
V	0x56
W	0x57
X	0x58
Y	0x59
Z	0x5A
VK_LWIN	0x5B
VK_RWIN	0x5C
VK_APPS	0x5D
VK_SLEEP	0x5F
VK_NUMPAD0	0x60
VK_NUMPAD1	0x61
VK_NUMPAD2	0x62
VK_NUMPAD3	0x63
VK_NUMPAD4	0x64
VK_NUMPAD5	0x65
VK_NUMPAD6	0x66
VK_NUMPAD7	0x67
VK_NUMPAD8	0x68
VK_NUMPAD9	0x69
VK_MULTIPLY	0x6A
VK_ADD	0x6B
VK_SEPARATOR	0x6C
VK_SUBTRACT	0x6D
VK_DECIMAL	0x6E
VK_DIVIDE	0x6F
VK_F1	0x70
VK_F2	0x71
VK_F3	0x72
VK_F4	0x73
VK_F5	0x74
VK_F6	0x75
VK_F7	0x76
VK_F8	0x77
VK_F9	0x78
VK_F10	0x79
VK_F11	0x7A
VK_F12	0x7B
VK_F13	0x7C
VK_F14	0x7D
VK_F15	0x7E
VK_F16	0x7F

Symbolic Name	Value
VK_F17	0x80
VK_F18	0x81
VK_F19	0x82
VK_F20	0x83
VK_F21	0x84
VK_F22	0x85
VK_F23	0x86
VK_F24	0x87
VK_NUMLOCK	0x90
VK_SCROLL	0x91
VK_LSHIFT	0xA0
VK_RSHIFT	0xA1
VK_LCONTROL	0xA2
VK_RCONTROL	0xA3
VK_LMENU	0xA4
VK_RMENU	0xA5
VK_BROWSER_BACK	0xA6
VK_BROWSER_FORWARD	0xA7
VK_BROWSER_REFRESH	0xA8
VK_BROWSER_STOP	0xA9
VK_BROWSER_SEARCH	0xAA
VK_BROWSER_FAVORITES	0xAB
VK_BROWSER_HOME	0xAC
VK_VOLUME_MUTE	0xAD
VK_VOLUME_DOWN	0xAE
VK_VOLUME_UP	0xAF
VK_MEDIA_NEXT_TRACK	0xB0
VK_MEDIA_PREV_TRACK	0xB1
VK_MEDIA_STOP	0xB2
VK_MEDIA_PLAY_PAUSE	0xB3
VK_LAUNCH_MAIL	0xB4
VK_LAUNCH_MEDIA_SELECT	0xB5
VK_LAUNCH_APP1	0xB6
VK_LAUNCH_APP2	0xB7
VK_OEM_1	0xBA
VK_OEM_PLUS	0xBB
VK_OEM_COMMA	0xBC
VK_OEM_MINUS	0xBD
VK_OEM_PERIOD	0xBE
VK_OEM_2	0xBF
VK_OEM_3	0xC0

Symbolic Name	**Value**
VK_OEM_4	0xDB
VK_OEM_5	0xDC
VK_OEM_6	0xDD
VK_OEM_7	0xDE
VK_OEM_8	0xDF
VK_OEM_AX	0xE1
VK_OEM_102	0xE2
VK_PROCESSKEY	0xE5
VK_PACKET	0xE7
VK_DBE_ALPHANUMERIC	0xF0
VK_DBE_KATAKANA	0xF1
VK_DBE_HIRAGANA	0xF2
VK_DBE_SBCSCHAR	0xF3
VK_DBE_DBCSCHAR	0xF4
VK_DBE_ROMAN	0xF5
VK_ATTN	0xF6
VK_CRSEL	0xF7
VK_EXSEL	0xF8
VK_EREOF	0xF9
VK_PLAY	0xFA
VK_ZOOM	0xFB
VK_NONAME	0xFC
VK_PA1	0xFD
VK_OEM_CLEAR	0xFE

Glossary

Automation Objects
Objects internal to CETerm that provide access to device, application, and session features from the script engine.

CEBrowseX
A Naurtech ActiveX control which provides access to the CETerm Automation Objects from a Windows Mobile device.

external
This is the name of an internal object in the DOM of the Windows CE 5.0 browser that gives access to the CETerm Automation Objects.

IDA Action Code
An IDA Action Code defines a special device, application, or emulation action within the Naurtech clients. IDA codes can be tied to keys, or KeyBars, and invoked via META tags or JavaScript. See the Appendix for a list of values.

Index