Code Reader JavaScript Programming Guide

code

MASTER DATE: 2008-02-06
COGE:

Reviewed By	Role	Signature	Date
Ryan Hoobler	COGE		2008-02-06
Weiyang Zhou	Engineering	Via Email	2008-02-06
Kerri Anne Humpherys	Marketing		2008-02-06

NO WARRANTY. This technical documentation is provided AS-IS. Further, the documentation does not represent a commitment on the part of Code Corporation. Code Corporation does not warrant that it is accurate, complete or error free. Any use of the technical documentation is at the risk of the user. Code Corporation reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult Code Corporation to determine whether any such changes have been made. Code Corporation shall not be liable for technical or editorial errors or omissions contained herein; nor for incidental or consequential damages resulting from the furnishing, performance, or use of this material. Code Corporation does not assume any product liability arising out of or in connection with the application or use of any product or application described herein.

NO LICENSE. No license is granted, either by implication, estoppel, or otherwise under any intellectual property rights of Code Corporation. Any use of hardware, software and/or technology of Code Corporation is governed by its own agreement.

Code Corporation, 11814 S. Election Road, Suite 200, Draper, UT 84020

www.codecorp.com

Table of Contents

Table of Tables

Table of Figures

Changes from Last Release

Note: this table reflects only major changes since last release; there may be additional changes that are not listed here.

Description	**Section(s)**
Added property to storage.upload	4.3.1.7

1 Introduction

Code Corporation (Code) designs, develops and manufactures image-based readers and software tools for data collection applications. With expertise in software development, optics, imaging, and Bluetooth™ wireless technology, Code is an innovative leader in the Auto ID and Data Collection Industry.

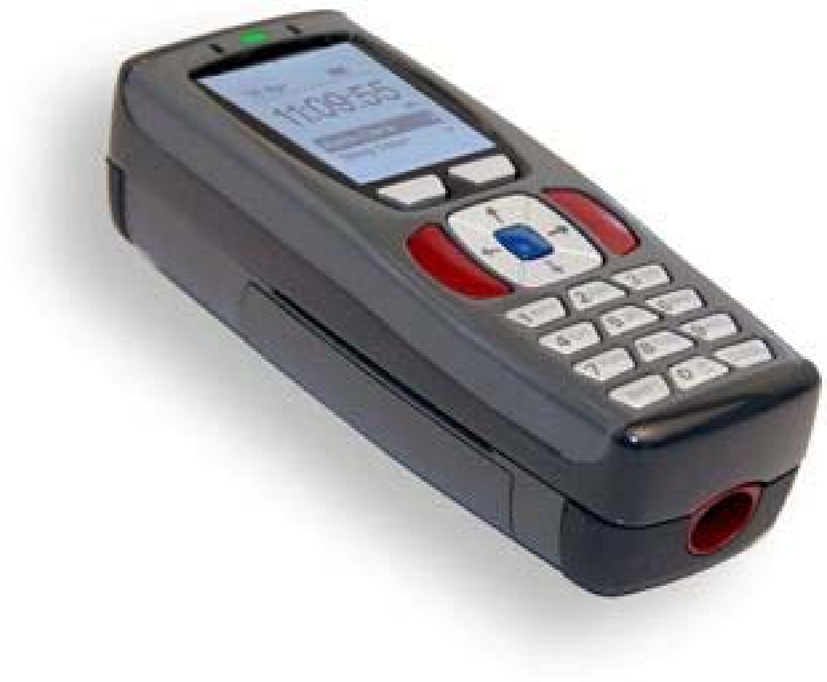


Figure 1 – Code Reader 3

The Code Reader 3 (CR3) is the smallest and lightest full-featured bar code reading terminal on the market. It combines all-symbology bar code reading with information display and keypad entry in an ergonomic hand-held platform. Code provides an easy-to-use JavaScript based application development interface for the CR3 as well as its other Code Readers (Code Reader 2 and Code Reader 1200). In order to run a custom JavaScript application, a JavaScript license may have to be purchased separately. Please contact your distributor or Code Corporation for additional information.

Table of Contents

1.1 Product Description

This manual describes the application programming interface for the Code Reader. It is assumed the reader will have programming skills and familiarity with the JavaScript language.

- Code Reader reads code data and can be programmed to transmit code data over a selected communications link or to store code data in Code Reader memory (batch mode).
- The programming environment provides interfaces to:
  - Read and manipulate data in Code Reader memory.
  - Display information on CR3 display screen.
  - Retrieve data from Code Reader hardware or CR3 key pad.
  - Access data sent by host.
  - Transmit data to a host computer via communications link.
  - Select type of communications link.
  - Set, change, and retrieve Code Reader configuration settings.

Table of Contents

1.2 Document Organization

This document is organized as follows:

- Section 1, Introduction: gives a product description and describes how to use this document.
- Section 2, Programming Environment: identifies tools used to create and load application software into Code Reader.
- Section 3, Programming Concepts: discusses how to accomplish various operations on the Code Reader using Code's application programming interface.
- Section 4, Class Reference: presents classes, objects, methods, properties, and constructors that support application programs.
- Glossary
- Appendices

Table of Contents

1.3 Document and Coding Conventions

This document employs the following conventions to aid in readability:

- Words that are part of the application development description use the `Courier New` font.
- Code examples use the **`bold Courier New`** font.
- Variable names that must be supplied by the programmer are `Courier New` font and are enclosed in relational signs, for example, `<variable_name>.`

The Code Reader JavaScript library uses the following naming conventions:

- identifiers**:** mixed-case with a capital letter where words join (soCalledCamelCase); acronyms and other initialisms are capitalized like words, e.g., nasaSpaceShuttle, httpServer, codeXml
- variables and properties: initial lower case
- classes (i.e., constructors): initial capital
- functions: initial lower case
- unit of measure: suffix to name, separated from name by underscore, using correct case when it's significant, e.g., offset_pixels, width_mm, power_MW, powerRatio_dB

Table of Contents

1.4 Related Documents

Code, *Code Reader 2,3 & 1200 – User Manuals*
Document Number C001537, C002332, and C003525

Code, *Code Reader 2/3/1200 – Reader-Host Interface Specification*
Document Number V002450

Code, *Reader Setup Utility User Manual*
Document Number C001720

Code, *CodeViewer User Manual*

Document Number C002507

Note: please visit Code's website at http://www.codecorp.com to obtain these documents.

Table of Contents

1.5 Related Utilities

Reader Download Utility (C002640) – Downloads JavaScript applications and data files from a host PC to the reader. Valid communication modes are USB Downloader, USB Virtual Com, and RS232.

Reader Setup Utility (V006807) – Provides a user interface to configure the reader. Valid communication modes are RS232 (57,600 Baud), and USB Virtual Com.

USB Virtual Com Driver (C002712)- Software driver that creates a virtual COM port for a USB-cabled reader so it can be used by a computer program that requires input from a serial device.

File Uploader (C002880) – Utility to transfer files from the reader to the host PC. Valid communication modes are RS232(115 Baud), USB Downlaoder mode, and USB Virtual Com.

These utilities are available at: http://www.codecorp.com/downloads.html

Table of Contents

2 Programming Environment

Code provides an environment for programming, testing, and loading Code Reader applications. JavaScript was selected as the programming language and Code implemented a Code Reader resident JavaScript engine.

Code provides a computer resident simulator and bundled editor for developing of Code Reader JavaScripts, which can be downloaded to the Code Reader.

Table of Contents

2.1 JavaScript Resources

This document is not a JavaScript manual. The following sources provide JavaScript reference books and online documents.

- ***JavaScript: The Complete Reference, Second Edition***
  by Thomas Powell, et al.
- ***JavaScript Demystified (Demystified)***
  by James Keogh.
- ***JavaScript (TM) in 10 Simple Steps or Less***
  by Arman Danesh.
- http://javascript.internet.com/
- http://www.javascript.com/

Table of Contents

2.2 Editor

You can use your favorite editing product to create and modify JavaScript code. Turn off any smart quote options in the editor. Smart quotes are not valid in JavaScripts.

In addition, Code has bundled a freeware editor (SciTE) with the Code Reader JavaScript Engine (JSE) Simulator. See Appendix A.

Table of Contents

2.3 Simulator

Code provides a Windows PC based simulator for JSE. See Appendix A for more information.

Table of Contents

2.4 CR3 CodeViewer Application

The CR3 CodeViewer Application runs as a JavaScript application on the CR3. The menu driven application has features for changing CR3 configuration settings and for defining the applications that run on the CR3. JavaScript Developers can make use of the following keywords in the CodeViewer Application:

`Title` – Displays the title of the JavaScript rather than the file name in CodeViewer's 'Application' menu. Add a comment to your script formatted as `$Title: <title of script>$` to implement.

`Revision` – Displays the revision of the JavaScript from the CodeViewer's 'Application/<script>' submenu. Add a comment to your script formatted as `$Revision: <revision of script>$` to implement.

For details about the utility, see *CodeViewer – User Manual,* Code Document Number C002507.

Table of Contents

2.5 Security

Code supplies an encryption utility for license protection.

- Each Code Reader contains a unique reader ID.
- Selected features of the Code Reader are protected by license.
- Code provides a license file that activates protected features purchased by the customer.
- A license file is required for each Code Reader licensed to use protected features.
- Third party software licenses may also be protected using the encryption utility.

Table of Contents

2.6 Debugging

The Code Reader contains a built-in error log that can be useful when debugging scripts. To debug the script when an error has occurred on the reader, issue the '(' command to the reader; the reader responds by sending the error log to the communications port. The error log may contain messages from the firmware that should be ignored. JavaScript errors in the log can be identified by the format: filename:lineNumber. If there are many error codes in the error log, you can issue the ‘)’ command to clear the log and repeat the steps to create the error, leaving only one entry in the log.

Example:

Error log returns:

```
X-ap/gerror-log. storage_init: flMountVolume fail status 26,
formatting.storage_formatFilesystem: status 0.temp.js:3: .

TypeError: gui.aler is not a function. X-ap/dEOF.
```

This error log contains one firmware error and one JavaScript error. The JavaScript error description begins with temp.js:3: and tells us that on line three of the temp.js file, gui.aler is not recognized as a function. In this case, gui.alert has been misspelled (it is missing the t).

Table of Contents

3 Programming Concepts

To help the developer create unique applications for the Code Reader, Code provides an easy to use, object oriented Code Reader JavaScript application programming interface. The application developer can create complex business applications with simple prompts and simple data entry through the CR3 user interface features (keypad and display screen).

The features of the programming interface include:

- Simplicity
- A graphical user interface
- Event handlers
- Symbol decoding
- Host communications
- Local data storage
- Code Reader configuration

In support of these features, the environment defines the following objects:

- `gui`
- `reader`
- `storage`
- `comm`

Using these objects and their methods and properties, you can create robust, interactive, and sophisticated user applications.

Code provides a CR3 JavaScript Simulator (Appendix A) for testing scripts and a Download Utility (section 1.5) for transferring scripts to the Code Reader.

A script can be made the default application using the configuration utility, or it may be run from the configuration utility without making it the default.

Note: the default application supplied by Code allows scripts to be run by host command or configuration code scan; the command is “|run:scriptName.js” (using your own scriptName).

Table of Contents

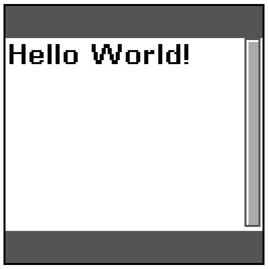
3.1 Simplicity

The “Hello World!” application is traditionally the first application presented in a programming guide. It is an easy to code and understand application that illustrates how the programming environment works.

In its simplest form the “Hello World!” application in the CR3 environment sends text to the display. With the following single line of code, you can display “Hello World!” in the screen defined by the standard CR3 `gui` object (section 4.1).

```
gui.show(new gui.Text("Hello World!"));
```

Execution of this script displays the image shown in Figure 2.



Hello World!

Figure 2 – Hello World Application

Note that in Figure 2, the text is displayed in a text box control with a scroll bar to the right as defined by the CR3 `gui` object.

Table of Contents

3.2 The CR3 `gui` Object

The CR3 application development environment defines a standard GUI display for application software (Figure 3). The display supports simple prompts and data entry.

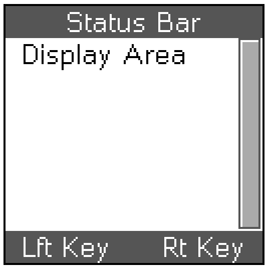


Figure 3 – The Standard `GUI` Display

The standard display consists of a status bar, a display area, and labels for the left and right software programmable keys (softkeys) at the top of the CR3 key pad (see Figure 7).

The scroll bar on the right side of the screen indicates the relative position within the displayed object as the operator scrolls through forms, menus, or text using the up and down keys on the keypad. This scrolling feature allows the application to display objects larger than the display area.

You can use the CR3 features of the gui object (section 4.1) to develop form and menu applications. Use the gui constructors to build forms and menus and the “show” methods to display them.

Table of Contents

3.2.1 Softkey Implementation

Softkeys are general purpose, programmable keys. If defined, the softkeys are always active in that they are not associated with or dependent on any control in the GUI display. Pressing the key always calls the event handler. The gui.showForm, gui.showMenu, and gui.showSubmenu methods include softkey definitions appropriate for the implementation.

The following example shows the basic approach to labeling the softkeys and implementing their event handlers.

```
// define send-key functions used by common softkeys
function sendEnter()  { gui.sendKey(gui.key.enter); }
function sendEscape() { gui.sendKey(gui.key.escape); }

// create some common softkeys
var selectSoftkey = new gui.Softkey("Select", sendEnter);
var okSoftkey     = new gui.Softkey("OK",     sendEnter);
var backSoftkey   = new gui.Softkey("Back",   sendEscape);
var cancelSoftkey = new gui.Softkey("Cancel", sendEscape);
```

Code provides equivalent definitions to the example above as a part of the CR3 JavaScript Library gui object. These examples have been provided so you can use them to create your own softkey definitions.

Table of Contents

3.2.2 Forms

Use the gui.Form object (section 4.1.3.3) to define the forms for your application. Section 4.1.3 defines the form object and several constructors that you can use to create controls on your application form.

The following code example demonstrates how to build and display a form. The event handler functions are empty in the example. The application developer defines the processing within the handlers.

```
// JavaScript Form Demo Script Document
// form event handlers
function myFormOnOk(){/* processing code */}

function myFormOnCancel(){/* processing code*/}
```

```
// create the form object
var myForm = new gui.Form(myFormOnOk, myFormOnCancel);

// create the edit control
var edit = new gui.Edit("");

// create the label control
var label = new gui.Label("Employee #:");

// position the controls on the form
myForm.append(label);
myForm.append(edit);

//Specify a child to be active when the form is displayed
(optional)
myForm.setActiveChild(edit);

// Create the caption that will appear on the status bar
myForm.caption = "form demo";

// show the form
gui.showForm(myForm);
```

When the Form Demo Script runs, the CR3 displays the following image:

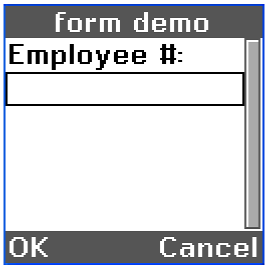


Figure 4 – Form Demo Display

The operator keys an employee number into the edit control and presses the left softkey (OK) to submit the entry to the application.

Table of Contents

3.2.3 Menus

Use the `gui.Menu` object (section 4.1.3.6) to define the menus for your application. Use the `gui.MenuItem` constructor to define the controls in the menu. Each control has an associated `onClick` property that defines the function to receive control when the CR3 firmware detects an enter-key event.

The following code example demonstrates how to build and display menus and submenus. The event handler functions are empty in the example. The application developer defines the processing within the handlers.

```
// JavaScript Menu Demo Script Document
// menu event handlers
function onTimeCard(){/* processing code */
                      alert("TimeCard");}
function onInventory()
{
    gui.showSubMenu(subMenu, myMenu);
}

function onCapital(){alert("capital");}
function onStock(){alert("stock");}

// create menu objects
var myMenu = new gui.Menu();
var subMenu = new gui.Menu();

// create menu entries
var timeCardApp =
    new gui.MenuItem("Time Card", onTimeCard);
var inventoryApp =
    new gui.MenuItem("Inventory", onInventory);
var separator =
    new gui.Separator(1, gui.separatorStyle.horizontalLine);
myMenu.caption = "menu demo";
subMenu.caption = "subMenu demo";

// create subMenu entries
var capital =
    new gui.MenuItem("Capital", onCapital);
var stock =
    new gui.MenuItem("Stock", onStock);

// position the controls on the menus
myMenu.append(separator);
myMenu.append(inventoryApp);
myMenu.append(timeCardApp);

subMenu.append(capital);
subMenu.append(stock);

//Specify a child to be selected when the menu is displayed
(optional)
myMenu.setActiveChild(inventoryApp);
subMenu.setActiveChild(capital);

// set the caption text for the status bar
myMenu.caption = "menu demo";
```

```
// show the menu
gui.showMenu(myMenu);
```

When the Menu Demo Script runs, the CR3 displays the following image:

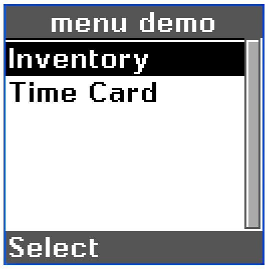


Figure 5 – Menu Demo Display

The `Select` softkey sends `gui.softkey.enter` to run the selected (highlighted) application. In this example, the Inventory menu item (a submenu). The script then displays the Inventory submenu shown in Figure 6.

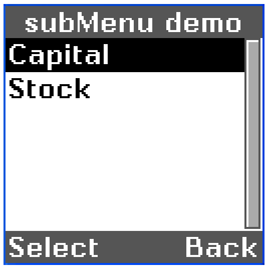


Figure 6 – Sub Menu Demo Display

Table of Contents

3.2.4 Text

Use the `gui.Text` object (section 4.1.3.11) to show text in the display area of the `gui` object. Do not use it as a text control within a menu or form. Text can exceed the capacity of the display area. The operator can scroll through the data using the up and down arrow keys on the CR3 keypad.

Table of Contents

3.3 Event Handlers

The Code Reader JavaScript environment is event-driven. The Code Reader firmware waits for an event such as a pressed key. The application gains control of an event by setting an object's event handler properties to functions. Events include:

- send and receive of communications packets
- decode operations
- pressed keys
- command execution
- change of reader mode (idle, standby, and power down)

An application gains control only when:

- The Code Reader application development environment defines an event handler property.
- The application creates an event handler function and assigns it to the event handler property.
- The event occurs and is not consumed.

The application can disable an event handler by setting the event handler property to null.

Table of Contents

3.3.1 Decode Event Handler

The `reader` object defines an event handler, `onDecode`. If set, the Code Reader calls the specified event handler with the decode results as the only handler parameter. Section 4.2.2.4 discusses decode event handling.

Example:

```
var numDecodes = 0;
var numDecodesProcessed = 0;

reader.onDecodeAttempt = function(count)
{
    numDecodes = count;
    numDecodesProcessed = 0;
}

reader.onDecode = function(decode)
{
    if( ++numDecodesProcessed < numDecodes )
    {
        // process individual decode, save in variables, etc.
    }
    else
    {
        // process the whole set, using saved variables, etc.
    }
}
```

Note: it is necessary to provide `onDecodeAttempt` only if you are interested in how decodes were grouped in an attempt or want to provide feedback on an attempt with zero decodes.

Table of Contents

3.3.2 Key Event Handling

The CLEAR, enter, and software programmable keys (softkeys) have special event handling capabilities in the CR3 application development environment. The event handlers are specified by various objects as constructor function parameters or as object properties.

Figure 7 contains a picture of the CR3 keypad.

The relationships between objects, keys, and event handler specifications are presented in Table 1. Three objects (`gui.Form`, `gui.Menu`, and `gui.Text`) have a constructor parameter to define event handlers for any non-consumed key. The GUI objects are documented in section 4.1.3.

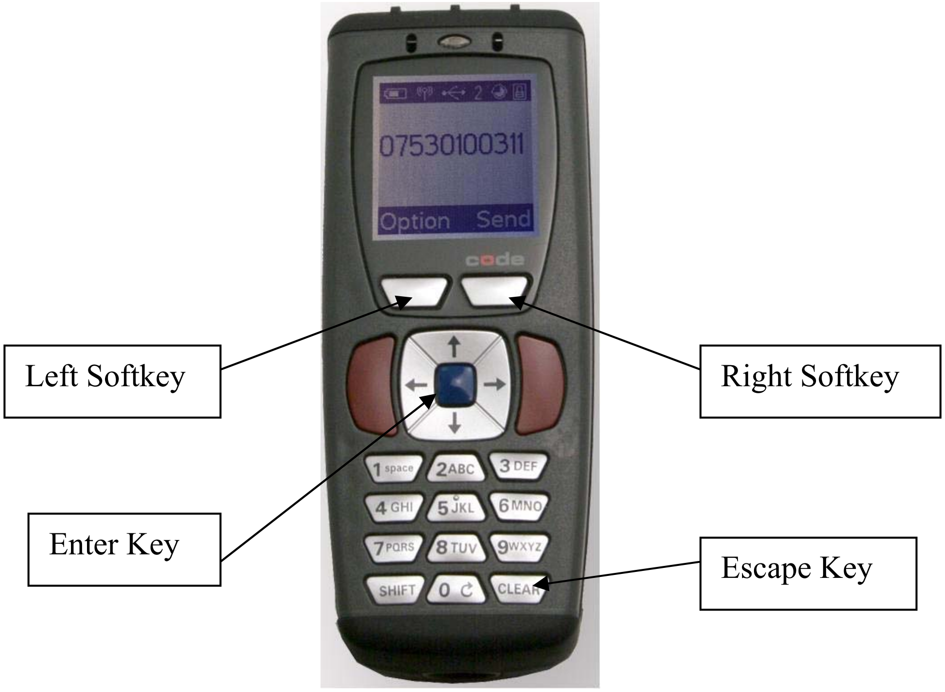


Figure 7 – CR3 Keypad

Table 1 – Key to Event Handler Mapping

Key	Object	Event Handler Property
Enter – The blue key at the center of the arrow keys	gui.Form gui.Menu gui.Text gui.Button gui.MenuItem	onOk onOk onOk onClick onClick
CLEAR – bottom right key (Note: clear is consumed by gui.Edit. A virtual Escape key, sent by cancelSoftkey and backSoftkey, also invokes onCancel and is not consumed by gui.Edit.)	gui.Form gui.Menu gui.Text	onCancel onCancel onCancel
Left softkey – upper left key	gui	onClick
Right softkey – upper right key	gui	onClick
Any non-consumed key	gui.Form gui.Menu gui.Text	onKey onKey onKey

Table of Contents

3.3.3 Command Execution

The Code Reader application environment defines a number of commands that can be sent to the Code Reader firmware from the host or by reading codes. The reader object (section 4.2) defines an event handler, onCommand. If onCommand is set, the Code Reader calls the specified event handler before execution of the command and passes the command type and command data to the handler through the calling parameters. The event handler can suppress or execute the received command by returning false or true respectively.

To receive control after a command has been completed, code the onCommandFinish (section 4.2.2.3) event handler.

Table of Contents

3.4 Symbol Decoding

The principal use of the Code Reader is capturing, decoding, and processing one-dimensional and two-dimensional bar codes. The Code Reader can read a wide range of code types, or symbologies, and provide access to the data after decoding using the `reader` object (section 4.2). The Code Reader decodes in response to:

- Pressing the read key on the key pad.
- A decode command from the `reader.processCommand` method.

The `reader.onDecode` property defines an event handler that allows the application to access the decode.

To implement a decode event handler, the application defines a handler function and sets it as the `onDecode` property of the `reader` object. Decode information is passed to the event handler as the first argument to `onDecode` (the argument properties are described in section 4.2.2.4).

```
function onDecode(decode)
{
    // Processing
}
reader.onDecode = onDecode;
```

There are four basic application options for processing a decode:

1. Process the data in the script, such as fill in form fields, and then consume the decode by coding `return null`.
2. Let the data be further processed by the Code Reader firmware, typically for sending and/or storing, by coding `return decode`.
3. Transform the data and let the Code Reader firmware process the changed data by setting `decode.data` as necessary and coding `return decode`.
4. Invalidate the decode by coding `return false`. The Code Reader will act as though the decode never occurred.

The following pseudo code presents an example of decode processing addressing the four options. The example transforms decode data based on certain symbologies. Then the example checks the format of the decode data to determine the next processing steps.

Subsections of this section discuss the processing steps in the following example.

Example:

```
function onDecode(decode)
{
    data = decode.data;

    if (decode.symbology == some-special-symbology)
    {
        data = transformed decode.data;
    }
    else if (decode.symbology
                == some-other-special-symbology)
    {
        data = differently transformed decode.data;
    }

    if (data matches employee-badge format)
    {
        loginForm.employeeField.text = decode.data;
        loginForm.pinField.text = "";
        gui.showForm(loginForm);
        return null;
    }
    else if (data matches part-number format)
    {
        stockForm.partField.text = decode.data;
        gui.showForm(stockForm);
        return null;
    }
    else if (data matches shelf-number format)
    {
        stockForm.shelfField.text = decode.data;
        gui.showForm(stockForm);
        return null;
    }
    else if (data matches wrong formats)
    {
        warning.text = "bad code for this application";
        gui.showForm(warning);
        return null;
    }
    else if (data matches format that is to be ignored)
    {
        return false;  // invalidate the decode
    }
    else // code should be processed by Code Reader firmware
    {
        if ( code should be processed
              with transformed data)
        {
           decode.data = data; // replace the data field
                              // with transformed data
        }
        return decode;
    }
}
```

Table of Contents

3.4.1 Transform Data by Symbology

Bar codes read by the Code Reader are encoded in unique symbologies. Particularly within two-dimensional codes, common data items may be present in different locations within the decode depending on the encoding symbology. In the example, line 5 checks the value of `decode.symbology` and transforms the decode data to a common format. To check symbology, compare `decode.symbology` against the symbology codes documented in *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

Note: Sometimes symbology is used to distinguish otherwise like-formatted data; for example, shelf tags may have the same number of digits as UPC codes for the products on the shelves, but have different bar code symbologies that can be used to determine whether the decode is a shelf tag or a product UPC code.

Table of Contents

3.4.2 Evaluate Data Format

After the data is converted into a common data format based on the symbology, the application determines the data format and processes according to data content.

```
if (data matches employee-badge format)
{
    loginForm.employeeField.text = decode.data;
    loginForm.pinField.text = "";
    gui.showForm(loginForm);
    return null;
}
else if data matches part-number format
{
    stockForm.partField.text = decode.data;
    gui.showForm(stockForm);
    return null;
}
else if (data matches shelf-number format)
{
    stockForm.shelfField.text = decode.data;
    gui.showForm(stockForm);
    return null;
}
```

The previous statements from the example demonstrate the processing of data within the decode handler. Based on the data format, the application program extracts data from the decode and displays appropriate forms.

These examples execute a `return null` statement to consume the decode for the specified data formats.

Table of Contents

3.4.3 Detect Format Errors

If the format matches a known format that should not be used in the current application context, the application can send a warning message, which is displayed in "warning" form.

```
else if data matches wrong formats
{
    warning.text = "bad code for this application";
    gui.showForm(warning);
    return null;
}
```

In this case, the example returns a `null` to consume the decode.

Note: Do not code `alert`, `confirm`, or `prompt`, either as functions or as gui methods, in an onDecode or onCommand event handler. The events originate in the Code Reader firmware, resulting from decodes, commands, or communication events. While the event handler is running, the main application is held idle until the event handler returns. If the event handler is waiting for the user to finish with `alert`, `confirm`, or `prompt`, the main application will be forced to wait as well, resulting in timeout errors.

Table of Contents

3.4.4 Let the Code Reader Process the Decode

If you want the Code Reader to process the decode, set the decode as the return statement parameter. If you have changed decode data and want the changes available to the Code Reader, set the appropriate data field in the decode to the changed value before returning the decode.

```
else // code should be processed by Code Reader firmware
{
    if ( code should be processed
          with transformed data)
    {
       decode.data = data; // replace the data field
                           // with transformed data
    }
    return decode;
}
```

Table of Contents

3.4.5 Ignore the Decode

You can ignore a particular format by exiting the function with a return value of `false` as shown in the following code segment from the example.

```
else if (data matches format that is to be ignored)
{
    return false;  // invalidate the decode
}
```

Note: Normally, the Code Reader will sound a good-decode beep at the end of decode processing. If you do not want invalidated decodes to cause the usual good-decode beep in the Code Reader firmware, you must configure the reader to process the decodes via JavaScript *before* beeping. Then the Code Reader will only beep if there is at least one decode that is not invalidated. See *Code Reader 2,3,& 1200 – Reader-Host Interface Specification*, Code document number V002450, specifically setting number 0x93.

If your `reader.onDecode` function returns `false`, you should configure the Code Reader beep this way.

Table of Contents

3.4.6 Determine the Orientation of the Decode

You can determine the orientation of a code by using the bounds array. The bounds array has four elements that can be used to give the coordinates of the four corners of the code (the origin is the center of the decode field):

- (decode.bounds[0].x, decode.bounds[0].y) = coordinates of top right corner
- (decode.bounds[1].x, decode.bounds[1].y) = coordinates of top left corner
- (decode.bounds[2].x, decode.bounds[2].y) = coordinates of bottom left corner
- (decode.bounds[3].x, decode.bounds[3].y) = coordinates of bottom right corner

These designations (e.g. top left) refer to the corners of the symbol, *not* as it appears in a particular image, but rather as it appears (most often) in its symbology specification. For example, for Data Matrix, array element 2, which contains the coordinates of the bottom left vertex of the symbol boundary, will *always* be proximate to the intersection of the two lines which form the "∟" of the symbol, regardless of the actual orientation (or mirroring) of the symbol in the image submitted to SwiftDecoder.

In normal orientation, we would expect the signs of the coordinates to be:

- decode.bounds[0].x (-), decode.bounds[0].y (+)
- decode.bounds[1].x (-), decode.bounds[1].y (-)
- decode.bounds[2].x (+), decode.bounds[2].y (-)
- decode.bounds[3].x (+), decode.bounds[3].y (+)

A code that is not "right side up" could be rejected by exiting the function with a return value of `false` as shown in the following example.

```
if (decode.bounds[0].x > 0 && decode.bounds[0].y < 0 &&
decode.bounds[1].x > 0 && decode.bounds[1].y > 0 &&
decode.bounds[2].x < 0 && decode.bounds[2].y > 0 &&
decode.bounds[3].x < 0 && decode.bounds[3].y < 0)
{
    return false;  // invalidate the decode
}
```

Note: Normally, the Code Reader will sound a good-decode beep at the end of decode processing. If you do not want invalidated decodes to cause the usual good-decode beep in the Code Reader firmware; you must configure the reader to process the decodes via JavaScript *before* beeping. Then the Code Reader will only beep if there is at least one decode that is not invalidated. See *Code Reader 2,3, & 1200 – Reader-Host Interface Specification*, Code document number V002450, specifically setting number 0x93.

Table of Contents

3.5 Host Communication

The Code Reader application development environment defines a host communication `comm` object (section 4.4) to support communications with a host resident application. For example, the Download Utility (section 1.5) is a host resident utility that communicates with the Code Reader for downloading files to the Code Reader.

From the host computer's view, the Code Reader is a serial device accessible through a serial or USB port, or through Bluetooth Radio Frequency (RF) communications. Code Reader configuration settings define the active host communications port.

The Code Reader host communications implementation supports two basic styles of communication: raw text and packets. It also supports a set of native protocols.

The application program transfers data to the host by writing to the Code Reader host communications port using the methods defined by the Code Reader `comm` object (section 4.4). Applications gain access to data sent by the host by implementing `onCommand` (and optionally `onCommandFinish`) event handlers defined by the Code Reader's `reader` object properties (section 4.2) and parsing the "|" command, for example.

Example:

```
reader.onCommand = function(type, data)
{
    // intercept | command with app-data: prefix

    if( type == '|'  &&  data.match(/^app-data\:/) )
    {
        return false;  // Suppress the command
    }

    return true;
}
```

For a full discussion of host communications, see *Code Reader 2,3, & 1200 – Reader-Host Interface Specification*, Code Document Number V002450.

Table of Contents

3.6 Data in Code Reader Local Storage

The application development environment provides program access to Code Reader local storage through the `storage` object (section 4.3). Data is maintained in storage as named objects called files. The Download utility can transfer host data into a Code Reader file. The Code Reader application can also store data in files.

The name of a Code Reader file may be 1 - 200 printable ASCII characters.

Use the `erase` and `write` methods of the `storage` object to manage files. Use the `findFirst` and `findNext` methods to locate files. Use the `read` method to access a file or the `upload` method to send it to the host.

Table of Contents

3.7 Reader Configuration

The Code Reader configuration settings define the active capabilities of the Code Reader. The application development environment defines the `reader` object (section 4.2), which contains methods for manipulating Code Reader settings. The document *Code Reader 2,3, & 1200 – Reader-Host Interface Specification*, Code Document Number V002450, defines the configuration items and the values that can be set for each item.

The application developer can dynamically change the active settings by using the `reader.writeSetting` method. This method changes the operational value of the setting, but that value is lost when the reader is turned off. The current values of all settings can be saved by using the `reader.saveSettings` method, which writes the current values of the settings to flash memory from where they are restored on power up.

Example:

```
reader.writeSetting(0x1b, 4);
if ( gui.confirm("Setting changed.\n\nSave now? ",
                 "Setting Change") )
   if ( !reader.saveSettings() )
         alert("Error Saving Settings");
```

Retrieve the current value of a setting by using the `reader.readSetting` method. Restore factory default settings by using the `reader.defaultSettings` method.

Table of Contents

4 Class Reference

The built-in objects described in this section enable a JavaScript program to receive data from the Code Reader and control its behavior.

Table of Contents

4.1 gui

The `gui` object provides application programming access to the CR3 display screen. The CR3 application development environment defines a standard software GUI format (section 4.1.3) consisting of a status bar, a display area, and labels for the left and right software programmable keys (softkeys) on the CR3 key pad.

The properties, methods, and classes of the `gui` object support the development of graphical user interfaces in custom software applications.

Table of Contents

4.1.1 Methods

The following section documents the methods defined for the CR3 `gui` object.

Table of Contents

4.1.1.1 alert

The `gui.alert` function displays text in the display area of the standard GUI display. Do not call this function within onDecode and onCommand event handlers.

Format:

```
gui.alert(text, title);
```

Where:

`text` – string; text to display as the alert.

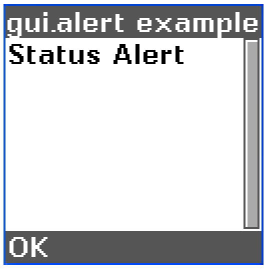
`title` – string; text to display in the gui object status bar; defaults to “Alert.”

Processing suspends until the operator presses an enter key – either the enter key or the left softkey defined as OK.

Example:

```
gui.alert("Status Alert", "gui.alert example");
```

Displays the alert shown in Figure 8 and waits until the operator presses the enter key or the left softkey (OK).



gui.alert example

Status Alert

Figure 8 – gui.alert Example

Table of Contents

4.1.1.2 confirm

The `gui.confirm` function displays text in the display area of the standard GUI display and returns a value based on the key pressed. Do not call this function within onDecode and onCommand event handlers.

Format:

```
result = gui.confirm(text, title,
                     leftSoftkeyLabel, rightSoftkeyLabel);
```

Where:

`text` – string; text to display for confirmation

`title` – string; text to display in the gui object status bar; defaults to “Confirm.”

`result` – Boolean; `true` if the confirm receives an enter key (either the enter key or the left softkey`)`; `false` if the confirm receives the right softkey. `leftSoftkeyLabel` – string; text to use as label for the left softkey (default is "Yes").

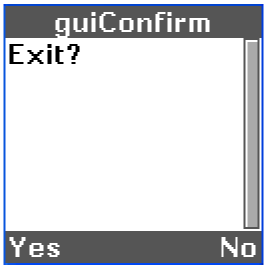
`rightSoftkeyLabel` – string; text to use as label for the right softkey (default is "No").

Processing suspends until the operator presses an enter key or cancel key.

Example:

```
while( !gui.confirm("Exit?", "guiConfirm") );
```

Displays the confirm dialog shown in Figure 9 and waits until the operator presses the enter key or the left softkey.



guiConfirm

Exit?

Figure 9 – gui.Confirm Example

Table of Contents

4.1.1.3 enableRefresh

The `gui.enableRefresh` function can be used to temporarily disable screen updates to speed up GUI object construction. The JavaScript application must reenable refresh after GUI object construction is completed.

Format:

```
gui.enableRefresh(enable);
```

Where:

`enable` – Boolean; true enables screen updates, false disables screen updates

Example:

```
gui.enableRefresh(false);

//build GUI objects

gui.enableRefresh(true);
```

Notes: Requires firmware 3478+

Table of Contents

4.1.1.4 prompt

The `gui.prompt` function displays text in the display area of the standard GUI display and returns a value based on the key pressed. Do not call this function within onDecode and onCommand event handlers.

Format:

```
result = gui.prompt(text, initial, title);
```

Where:

`text` – string; text to display as a label above a `gui.Edit` control

`initial` – string; the initial string to display as the contents of edit control; default is an empty string.

`title` – string; text to display in the gui object status bar; defaults to "Prompt".

`result` – string; contents of the edit control if the prompt receives an enter key (either the enter key or the left softkey defined as `OK`); null if the prompt receives the right softkey defined as `Cancel`.

Processing suspends until the operator presses an enter key or `Cancel` key. The operator can key new data into the edit control before pressing enter or the left softkey.

Example:

```
string = gui.prompt("Enter login ID", "None", "guiPrompt");
```

Displays the prompt shown in Figure 10 – gui.Prompt Example.

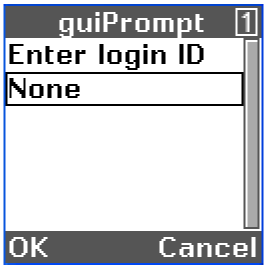


Figure 10 – gui.Prompt Example

The value of `string` depends on the operator action.

- If the operator presses the right softkey (Cancel), the value of `string` is `null`.
- If the operator presses the blue "enter" key or the left softkey (OK) the value of `string` is:
  - <new content> if the operator changes the contents of the edit control
  - "None" if the operator does not change the content.

Table of Contents

4.1.1.5 sendKey

The `gui.sendKey` method sends a "pressed key" indication to the CR3 firmware as though it came from CR3 keypad.

Format:

```
result = gui.sendKey(key);
```

Where:

`key` – number constant; the key to send. Use number constants defined in section 4.1.2.2.

`result` – Boolean; `true` if successful; `false` if not, which usually means the keypad is locked but can also mean that the key buffer is full.

Example:

```
gui.sendKey(enter);
```

Sends the enter key event to the CR3 firmware as though the operator had pressed the enter key.

Table of Contents

4.1.1.6 sendText

The `gui.sendText` method sends a text string to the CR3 `gui` object as though it had been entered from the keypad.

Format:

```
result = gui.sendText(text);
```

Where:

`text` – string; the text to send.

`result` – Boolean; `false` if all specified text could not be sent to the GUI (in which case, none of it will have been sent); otherwise, `true`.

Example:

```
reader.onDecode =
   function(decode) { gui.sendText(decode.data); }
```

Sends all decode data to the `gui` object as though it had been entered from the keypad.

Table of Contents

4.1.1.7 setFont

The `gui.setFont` method sets the font of the CR3 gui display to the specified font.

Format:

```
gui.setFont(font);
```

Where:

`font` – string; the file name of the font.

`result` – Boolean; `false` if the specified font file could not be loaded; otherwise, `true`.

Example:

```
gui.setFont("myFontFile.fnt");
```

Sets the GUI font to myFontFile.

Notes: The default font is uniread10Bold.fnt. Contact Code Corporation for a list of currently available fonts.

Table of Contents

4.1.1.8 show

The `gui.show` method instructs the CR3 to write the specified form, menu, or text object to the CR3 display as a standard `gui` object (section 4.1.3).

This low level approach is not recommended for use in most applications. Instead, Code recommends using the `gui.showForm`, `gui.showMenu`, and `gui.showSubMenu` methods.

Format:

```
gui.show(object);
```

Where:

`object` – object to show on the display. The object must be a `gui.Form` `gui.Menu`, or `gui.Text` object (section 4.1.3).

Note: This method does not return a value.

Table of Contents

4.1.1.9 showForm

The `gui.showForm` method instructs the CR3 to display the specified form on the CR3 display as a standard `gui` object (section 4.1.3).

Format:

```
gui.showForm(yourForm);
```

Where:

`yourForm` – form object to show on the display; the object must be a `gui.Form` object (section 4.1.3.3).

Note: This method does not return a value.

To insert a caption into the status bar, set the `yourForm.caption` property.

By default, the left software programmable key is set to `gui.okSoftkey` (section 4.1.4.3). You may also define a custom `leftSoftkey` for your `form` object, e.g., `yourForm.leftSoftkey = yourSoftkey`, in which case `gui.showForm` will use your softkey.

By default, the right software programmable key is set to `gui.cancelSoftkey` (section 4.1.4.2). You may also define a custom `rightSoftkey` for your `form` object.

Table of Contents

4.1.1.10 showMenu

The `gui.showMenu` method instructs the CR3 to display the specified menu on the CR3 display as a standard `gui` object (section 4.1.3). This menu is the top level menu; sub-menus can be created using the `gui.showSubMenu` method.

Format:

```
gui.showMenu(yourMenu);
```

Where:

`yourMenu` – menu object to show on the display. The object must be a `gui.Menu` object (section 4.1.3.6).

Note: This method does not return a value.

To insert a caption into the status bar, set the `yourMenu.caption` property.

This method sets the left software programmable key to `gui.selectSoftkey` (section 4.1.4.4).

This method sets the right software programmable key to `gui.backSoftkey` (section 4.1.4.1) if the `yourMenu.onCancel` property is set; otherwise, `null`.

Table of Contents

4.1.1.11 showSubMenu

The `gui.showSubMenu` method instructs the CR3 to display the specified menu on the CR3 display as a standard `gui` object (section 4.1.3).

Format:

```
gui.showSubMenu(yourMenu, parentMenu);
```

Where:

`yourMenu` – menu object to show on the display. The object must be a `gui.Menu` object (section 4.1.3.6).

`parentMenu` – parent menu to display in response to `gui.backSoftkey`.

Note: This method does not return a value.

To insert a caption into the status bar, set the `yourMenu.caption` property.

This method sets the left software programmable key to `gui.selectSoftkey` (section 4.1.4.4).

This method sets the right software programmable key to `gui.backSoftkey` (section 4.1.4.1) and sets the menu object's `onCancel` property to a function that shows the parent menu.

Table of Contents

4.1.1.12 splash

The `gui.splash` method displays an image on the CR3 screen and suspends execution until a key is pressed. An optional timeout value will restart execution after a specified time, if no key is pressed.

Note: The key that is pressed to cancel the splash screen is consumed. If a decode occurs while the splash screen is displayed, the splash screen will be dismissed as if a key had been pressed; the decode is not consumed.

Format:

```
result = gui.splash(imageName, stringText, timeout_ms);
```

Where:

`imageName` – string; the name of the image file to display (section 4.1.3.4).

stringText – string; the text string to be displayed below the image in the softkey area of the display.

`timeout_ms` – number; the number of microseconds to wait before timeout of the splash display.

`result` – Boolean; `true` if a key was pressed before timeout; `false` if a timeout occurred.

Example:

```
result = gui.splash(CorpLogo.img, "Version 1", 2000);
```

displays a corporate logo image and the text "Version 1" on the display for two seconds.

The CR3 supports only its native format, which uses the extension `.img`. The image must be 128x128 pixels (for splash screen only). Images are not cropped; they will either display in their entirety or will not display at all. The CR3 displays the image in four grayscale values from white to black.

Code provides a utility to convert standard .pgm format files to the CR3's native .img format (contact Code for more information http://www.codecorp.com). Several image conversion programs are available, commercially and as freeware, to convert other formats to .pgm files.

Table of Contents

4.1.1.13 translateDigitToCustom

The `gui.translateDigitToCustom` method changes the keypad input to a custom key map.

Format:

```
gui.translateDigitToCustom = function(digit, index);
```

Where:

`digit` – number of button on keypad (0-9).

`index` – index of custom key map.

Example:

```
var keyMap = new Array("0.!@#$%&*()","1 @","abc2ABC","def3DEF",
"ghi4GHI","jkl5JKL","mno6MNO","pqrs7PQRS","tuv8TUV","wxyz9WXYZ");

gui.translateDigitToCustom = function(digit, index)
{
    if (keyMap[digit])
    {
        return keyMap[digit][index % keyMap[digit].length];
    }
    else
    {
        return digit;
    }
}
```

returns the characters specified by the "keyMap" variable.

Note: Requires firmware 3478+.

Table of Contents

4.1.2 Properties

The following section documents the properties defined for the CR3 `gui` object.

Table of Contents

4.1.2.1 inputMode

The `gui.inputMode` object contains constants that define input modes for the CR3.

The constant definitions are:

```
gui.inputMode.numeric
gui.inputMode.caps
gui.inputMode.lower
gui.inputMode.latinCaps
gui.inputMode.latinLowerCase
gui.inputMode.symbols
```

The character sets defined for these modes are described in Appendix B.

Table of Contents

4.1.2.2 key

The `gui.key` property is a read-only object containing number constants specifying keys for use with the `gui.sendKey` method. The constants are named:

- `up`
- `down`
- `left`
- `right`
- `enter` (the blue key on the keypad)
- `back` (“CLEAR” on the keypad)
- `escape`
- `home`
- `end`
- `leftSoftkey`
- `rightSoftkey`

Constants `escape`, `home`, and `end` have no keypad counterpart.

Constants `leftSoftkey` and `rightSoftkey` represent the left and right software programmable keys on the CR3.

Table of Contents

4.1.2.3 leftSoftkey

The `gui.leftSoftkey` property identifies an event handler for the `onClick` property of a `gui.Softkey` object and the key label, associated with the left programmable key on the CR3. The application program defines a `gui.Softkey` object. See the example in section 3.2.1

Setting `gui.leftSoftkey` to null disassociates the softkey object from the property (removing the event handler and the softkey label).

When menus and forms are shown using the `gui.showMenu`, `gui.showSubMenu`, and `gui.showForm` methods, the `gui.leftSoftkey` property is set automatically.

Table of Contents

4.1.2.4 rightSoftkey

The `gui.rightSoftkey` property identifies an event handler for the `onClick` property of a `gui.Softkey` object and the key label, associated with the right programmable key on the CR3. The application program defines a `gui.Softkey` object. See the example in section 3.2.1.

Setting `gui.rightSoftkey` to null disassociates the softkey object from the property (removing the event handler and the softkey label).

When menus and forms are shown using the `gui.showMenu`, `gui.showSubMenu`, and `gui.showForm` methods, the `gui.rightSoftkey` property is set automatically.

Table of Contents

4.1.2.5 statusText

The `gui.statusText` property is a string that specifies text for display in the status bar at the top of a CR3 GUI screen. When `gui.status` is null, the CR3 displays status icons in the status bar. Note: The input mode icon will always be displayed in addition to the status text when an edit control is active.

With menus and forms, use the `caption` property (section 4.1.6.1) to automatically set `gui.statusText` when the menu or form is shown.

Table of Contents

4.1.3 Objects

The CR3 application development environment provides the user classes described in this section for use in building forms for the CR3 `gui` object. The instances of these classes are referred to as `controls` in this document.

Table of Contents

4.1.3.1 gui.Button

The `gui.Button` constructor creates a button control for a GUI form. The `onClick` event handler is called when the enter key on the CR3 keypad is pressed and the button control is active. Program the function to return Boolean `true` if the control's default processing of the key should continue. Otherwise, program the function to return `false;` the control will act as if not clicked.

Format:

```
var <button_name> =
   new gui.Button(text, onClick);
```

Where:

`<button_name>` – program-provided button control.

`text` – string; a label for the button. This property can be changed after the object is created.

`onClick` – function for handling the button click event. The CR3 calls this function when the operator presses the `OK` enter key on the CR3 keypad when the GUI button is the active control.

Example:

```
// button control event handler
function rFOnClick(){reader.writeSetting(0x1b, 4);}
function rs232OnClick(){reader.writeSetting(0x1b, 1);}

// create the form object
var myForm = new gui.Form();

// create the button
var rfButton = new gui.Button("RF Comm", RFOnClick);
var rs232Button = new gui.Button("RS232 Comm", RS232OnClick);

// position the controls on the form
myForm.append(rfButton);
myForm.append(rs232Button);

// Place text on the status bar
```

```
gui.statusText = "button demo";

// show the form
gui.showForm(myForm);
```

Displays the form shown in Figure 11.

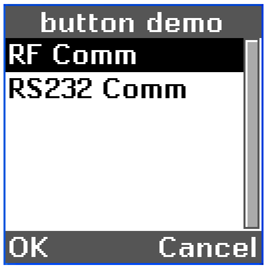


Figure 11 – Button Demo

When the operator presses the left softkey or the enter key when the control labeled “RF Comm” is active, the script executes a `reader.writeSettings` method to set the communications mode setting to RF (Bluetooth). When the “RS232 Comm” control is active and the operator presses the key, the script executes a `reader.writeSettings` method to set the communications mode setting to RS232.

Note: The active control is highlighted.

Table of Contents

4.1.3.2 gui.Edit

The `gui.Edit` constructor creates an edit control for a GUI form. The CR3 operator can enter data into the edit control.

Format:

```
var <edit_name> =
   new gui.Edit(text, defaultInputMode, validInputModes, onChar,
readOnly);
```

Where:

`<edit_name>` – program-provided edit control.

`text` – string; the initial value for the edit control. The control contains `text` when it is first displayed on the `gui` object. This property can be changed after the object is created.

`defaultInputMode` – number; the input mode that is selected when the user navigates to the edit control and enters data. Modes are defined by `gui.inputMode` (section 4.1.2.1).

Note: The user can change to another input mode using the shift key.

`validInputModes` – number; a bitwise combination of input modes as defined by `gui.inputMode` (section 4.1.2.1); defines the input modes that are valid in the edit control.

`onChar` – function; the function to run when a character is entered into an edit control.

`readOnly` – Boolean; false allows the text to be changed by the user, true prevents the text from being changed.

Example:

```
function quit() { reader.runScript(".default.js"); }

var form = new gui.Form(null, quit);
form.Caption = "Input Modes";

form.append(new gui.Edit("Num, shft any",
                          gui.inputMode.numeric));
form.append(new gui.Edit("CAP, shft any",
                          gui.inputMode.caps));
form.append(new gui.Edit("Num only",
                          gui.inputMode.numeric,
                          gui.inputMode.numeric));
form.append(new gui.Edit("CAP, U/l Case",
                          gui.inputMode.caps,
                          gui.inputMode.caps
                          | gui.inputMode.lowerCase));
gui.showForm(form);
```

Displays the form shown in Figure 12.

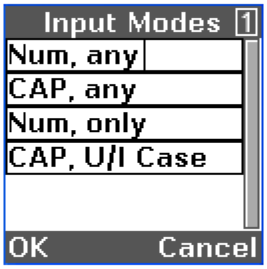


Figure 12 -- Input Modes Example

The text in each edit control identifies the default input mode of the control and the modes which are enabled for the shift key.

Note: onChar and readOnly require firmware 3478+

Table of Contents

4.1.3.3 gui.Form

The `gui.Form` constructor creates a `Form` object for the CR3 GUI. The `gui.Form` constructor defines three event handlers for key events. Event handlers are null if not specified.

The following controls can be used in a form:

- `gui.Button`
- `gui.ToggleButton`
- `gui.Edit`
- `gui.Image`
- `gui.Label`
- `gui.Separator`

Form controls must be appended (section 4.1.5.1) or prepended (section 4.1.5.2) to the form object.

Format:

```
var <form_name> = new gui.Form(onOk, onCancel, onKey);
```

Where:

`<form_name>` – program-provided form control.

`onOk` – function for handling the enter key. The CR3 calls this function when the operator presses the enter key on the CR3 keypad and the active control is not a button.

`onCancel` – function for handling the `CLEAR` key. The CR3 calls this function when the operator presses the key on the CR3 keypad and the active control is not an `edit` control. This function is also called when the escape key is issued as a softkey.

`onKey` – function for handling any key, soft or real, not consumed by the active control (section 4.1.6.2).

To add a label to the form in the status area, set the form's `caption` property to a string containing the label.

Example:

See section 3.2.2.

Table of Contents

4.1.3.4 gui.Image

The `gui.Image` constructor creates an image object that can be displayed in the CR3 GUI form.

Format:

```
var <image_name> = new gui.image(name);
```

Where:

`<image_name>` – program-provided image control.

`name` – string; the name of an image file in file storage (section 4.1.3.4).

Example:

```
var myForm = new gui.Form();
var image = new gui.Image("MyImage.img");
myForm.append(image);
gui.showForm(myForm);
```

The image can be up to 128x94 pixels depending on the form. Images are not cropped; they either display in their entirety or do not display at all. The CR3 displays the image in four grayscale values from white to black.

The image file format is specific to the CR3. Code provides a utility to convert standard .pgm format files to the CR3 native .img format. Several image conversion programs are available, commercially and as freeware, to convert other formats to .pgm files.

Table of Contents

4.1.3.5 gui.Label

The `gui.Label` constructor creates a label control that can be displayed in the CR3 GUI menu or form.

Format:

```
var <label_name> = new gui.Label(text);
```

Where:

`<label_name>` – program-provided label control.

`text` – string; the text to be displayed as a label. This property can be changed after the object is created.

Example:

See the form example in section 3.2.2.

Table of Contents

4.1.3.6 gui.Menu

The `gui.Menu` constructor creates a menu object for the CR3 GUI. The `gui.Menu` constructor defines three event handlers for key events. Event handlers are null if not specified.

The following controls can be used in a menu:

- `gui.MenuItem`
- `gui.Separator`
- `gui.ToggleButton`

Menu controls must be appended (section 4.1.5.1) or prepended (section 4.1.5.2) to the menu object.

Format:

```
var <menu_name> = new gui.Menu(onOk, onCancel, onKey);
```

Where:

`<menu_name>` – program-provided menu.

`onOk` – function for handling the enter key. The CR3 calls this function when the operator presses the enter key on the CR3 keypad when the active control is not a button.

`onCancel` – function for handling the `CLEAR` key. The CR3 calls this function when the operator presses the `CLEAR` key on the CR3 keypad and the active control is not an `edit` control. This function also is called when the escape virtual key is issued (typically by a softkey).

`onKey` – function for handling any key, soft or real, not consumed by the active control (section 4.1.6.2).

Example:

See the menus example in section 3.2.3.

Table of Contents

4.1.3.7 gui.MenuItem

The `gui.MenuItem` constructor creates a `MenuItem` control for display in a CR3 GUI menu. The `onClick` processing function is called when the enter key on the CR3 keypad is pressed and the `MenuItem` control is active.

Format:

```
var <menuItemItem_name> =
    new gui.MenuItem(text, onClick);
```

Where:

`<menuItem_name>` – program-provided `MenuItem` control.

`text` – string; a label for the `MenuItem`.

`onClick` – function for handling the `MenuItem`. The CR3 calls this function when the operator presses the enter key on the CR3 keypad when the `MenuItem` is the active control. Code the function to return Boolean `true` if the control's default processing of the key should continue. Otherwise, code the function to return `false`; the control will act as if not clicked.

Example:

See section 3.2.3.

Table of Contents

4.1.3.8 gui.MultiLineEdit

The `gui.MultiLineEdit` constructor creates a multiple line edit control for the GUI screen. The CR3 operator can enter data into the multiple line edit control. The `gui.MultiLineEdit` constructor consumes the entire GUI screen, so it cannot be appended/prepended to a menu or form. To access a multiple line edit control from a menu

Format:

```
var <multiLineEdit_name> =
  new gui.MultiLineEdit(text, defaultInputMode, validInputModes,
onChar);
```

Where:

`<edit_name>` – program-provided multiple line edit control.

`text` – string; the initial value for the multiple line edit control. The control contains `text` when it is first displayed on the `gui` screen. This property can be changed after the object is created.

`defaultInputMode` – number; the input mode that is selected when the user navigates to the edit control and enters data. Modes are defined by `gui.inputMode` (section 4.1.2.1).

Note: The user can change to another input mode using the shift key.

`validInputModes` – number; a bitwise combination of input modes as defined by `gui.inputMode` (section 4.1.2.1); defines the input modes that are valid in the edit control.

`onChar` – function; the function to run when a character is entered into a multiple line edit control.

Other Functionality:

`insert` – function, arg: string; this function inserts a string where the cursor is when the function is called.

Format: <multiLineEditControlName>.insert(string);

Where

`< multiLineEditControlName>` – program- provided multiple line edit control.

`string` – string; text to insert into multiLineEdit control.

Example:

```
var main = new gui.Menu
main.append(new gui.Button("Notes", function() {
gui.showDialog(captureNotes); }));

gui.showMenu(main);

storage.write("saveNotes.txt","");

var captureNotes = new gui.MultiLineEdit("", gui.inputMode.caps)
captureNotes.leftSoftkey = new gui.Softkey("Save", function()
{storage.append("saveNotes.txt", captureNotes.text);
captureNotes.text = ""; gui.showMenu(main); });
captureNotes.rightSoftkey = new gui.Softkey("Cancel", function()
{ captureNotes.text = ""; gui.showMenu(main); });
```

Note: Requires firmware 3478+

Table of Contents

4.1.3.9 gui.Separator

The `gui.Separator` constructor creates a separator control for display in a CR3 GUI menu or form. Use the separator to insert white space or lines into a form to increase separation between controls.

Format:

```
var <separator_name> =
   new gui.Separator(height, style);
```

Where:

`<separator_name>` – program-provided separator control.

`height` – number; the height in pixels of the separator; minimum 1 pixel.

`style` – number; the style of the separator. `style` must be selected from one of the following numeric constants:

- `gui.separatorStyle.blank`
- `gui.separatorStyle.horizontalLine`
- gui.separatorStyle.horizontalGroove
- gui.separatorStyle.horizontalRidge

The `gui.separatorStyle.horizontalLine` style adds a line in the approximate center of the separator space as shown in Figure 13.

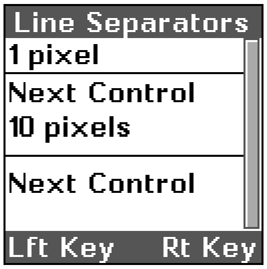


Figure 13 – gui.Separator Lines

Example:

See the menu example in section 3.2.3.

Table of Contents

4.1.3.10 gui.Softkey

The `gui.Softkey` object provides processing control of the programmable or “soft” keys on the CR3 just below the display screen.

Format:

```
var <softkey> = new gui.Softkey(text, onClick);
```

Where:

`<softkey>` – program-provided softkey object.

`text` – string; a label for the softkey; displays on the GUI.

`onClick` – function; the function to be executed when the softkey is pressed.

Set the `gui.leftSoftkey` or `gui.rightSoftkey` property to `<softkey>` as appropriate. The CR3 JavaScript Library defines a set of useful softkey objects (section 4.1.4).

Example:

```
function leftSoftkeyOnClick()
{
   /* processing code */
}
function rightSoftkeyOnClick()
{
   /* processing code */
}

var left = new gui.Softkey("Ok", leftSoftkeyOnClick);
var right =
          new gui.Softkey("Cancel", rightSoftkeyOnClick);

gui.leftSoftkey = left;
gui.rightSoftkey = right;
```

Table of Contents

4.1.3.11 gui.Text

The `gui.Text` constructor creates a text object that can be displayed in the CR3 GUI display area. Text length can exceed the capacity of the display area. The `Text` control includes a scroll bar to indicate relative position within the text when the operator presses the up and down arrow keys.

Format:

```
var <text_name> =
   new gui.Text(text, onOk, onCancel, onKey);
```

Where:

`<text_name>` – program-provided text control.

`text` – string; text data to display on the CR3 GUI. To display multi-line text, insert the new-line character ("\n") in the text string. This property can be changed after the object is created.

`onOk` – function for handling the enter key. The CR3 calls this function when the operator presses the enter key on the CR3 keypad.

`onCancel` – function for handling the `CLEAR` key. The CR3 calls this function when the operator presses the `CLEAR` key on the CR3 keypad. This function also is called when the escape key is issued (typically by a softkey).

`onKey` – function for handling any key, soft or real, not consumed by the active control (section 4.1.6.2).

Note: The `gui.Text` constructor should be used only to display text, not as a control within a `gui.Form` or `gui.Menu`.

Other Functionality:

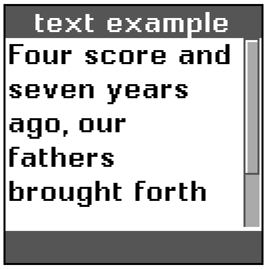
`leftClipString` – function with a single string argument.

Return: string; the input string parameter is truncated when necessary so the result string can be displayed without exceeding the maximum width of the CR3 display.

Example:

```
gui.statusText = "text example";
gui.show(new gui.Text
        ("Four score and seven years ago, our fathers brought
        forth upon this continent, etc ..."));
```

displays the screen shown in Figure 14.



text example
Four score and
seven years
ago, our
fathers
brought forth

Figure 14 – gui.Text Example

Note: The scroll bar indicates that there is more text to display than is currently on the screen.

Table of Contents

4.1.3.12 gui.ToggleButton

The `gui.ToggleButton` constructor defines a button control for a GUI form. When a toggle button is clicked, an indicator in the button is alternately displayed or suppressed.

Format:

```
var <togglebutton_name> =
   new gui.ToggleButton(text, initiallyChecked, onToggle);
```

Where:

`<togglebutton_name>` – program-provided toggle button control.

`text` – string; a label for the toggle button.

`initiallyChecked` – Boolean; `true`, the button displays the checked indicator when first shown; `false`, the button does not display the checked indicator when first shown.

onToggle – function for handling the button click event. It passes a single Boolean parameter; true, the button is checked; false, the button is not checked. The CR3 calls this function when the operator presses the OK enter key on the CR3 keypad when the GUI button is the active control.

Other Functionality:

checked – Boolean; current state of toggle button.

toggle – function; toggles the toggle button as if activated by the GUI screen.

Example:

```
// form event handlers
// button control event handler
function toggleOnClick(checked)
        {reader.writeSetting(0xa7, checked);}

// create the form object
var myForm = new gui.Form();

// create the button
var toggle =
    new gui.ToggleButton("Vibrate", false, toggleOnClick);

// position the controls on the form
myForm.append(toggle);

// Place text on the status bar
myForm.caption = "toggle demo";

// show the form
gui.showForm(myForm);
```

initially shows the form in Figure 15.

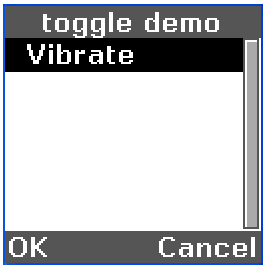
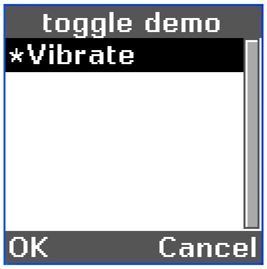


Figure 15 – Toggle Not Selected

Pressing the left softkey (OK) toggles the indicator, as shown in Figure 16, and turns on the vibrate feature of the CR3. Pressing OK again turns off the indicator and the vibrate feature.



toggle demo

*Vibrate

Figure 16 – Toggle Selected

Notes: toggleButton.checked and toggleButton.toggle() require firmware 3478+.

Table of Contents

4.1.4 Predefined Softkey Objects

The softkey objects described in this section are defined by the CR3 JavaScript library.

Table of Contents

4.1.4.1 backSoftkey

The `gui.backSoftkey` object defines a softkey object. It labels the softkey "Back" and sends the escape key when the softkey is clicked.

Example:

```
gui.rightSoftkey = gui.backSoftkey;
```

Table of Contents

4.1.4.2 cancelSoftkey

The `gui.cancelSoftkey` object defines a softkey object. It labels the softkey "Cancel" and sends the escape key when the softkey is clicked.

Format:

```
gui.rightSoftkey = gui.cancelSoftkey;
```

Table of Contents

4.1.4.3 okSoftkey

The `gui.okSoftkey` object defines a softkey object. It labels the softkey "OK" and sends the enter key when the softkey is clicked.

Format:

```
gui.leftSoftkey = gui.okSoftkey;
```

Table of Contents

4.1.4.4 selectSoftkey

The `gui.selectSoftkey` object defines a softkey object. It labels the softkey “Select” and sends the enter key when the softkey is clicked.

Example:

```
gui.leftSoftkey = gui.selectSoftkey;
```

Table of Contents

4.1.5 Form and Menu Common Methods

4.1.5.1 append(control)

The `append` function places the specified `control` as the last control in the specified menu or form.

Format:

```
<MenuOrForm_name>.append(control);
```

Where:

`control` – the control to append.

Note: A control cannot be used more than once in a form or menu.

Example:

See section 3.2.2.

Table of Contents

4.1.5.2 prepend(control)

The `prepend` function places the specified `control` as the first control in the specified menu or form.

Format:

```
<MenuOrForm_name>.prepend(control);
```

Where:

`control` – the control to `prepend` to the menu.

Note: A control cannot be used more than once in a menu or form.

Example:

See forms example in section 3.2.2.

Table of Contents

4.1.5.3 setActiveChild(control)

The `setActiveChild` selects (but does not activate) the specified control when the menu or form is displayed. This method is optional.

Format:

```
<MenuOrForm_name>.setActiveChild(control);
```

Where:

`control` – the control to `select` when the menu is displayed.

Example:

See forms example in section 3.2.2.

Note: You must show the form/menu after setting the active child in order for this function to work properly.

Table of Contents

4.1.6 Form and Menu Common Properties

The properties and methods described in the following section are common to the `gui.Menu` and `gui.Form` objects.

Table of Contents

4.1.6.1 caption

The `caption` property is a string that is used by `gui.showForm`, `gui.showMenu`, and `gui.showSubMenu` to display a caption in the status bar of the CR3 `gui` object.

Format:

```
<MenuOrForm_name>.caption = "<caption_string>";
```

Example:

See forms example in section 3.2.2.

Table of Contents

4.1.6.2 onKey

The `onKey` property is a property of type function that is used by `gui.Form`, `gui.Menu`, and `gui.Text` to provide control for any key not consumed by the active control. Key constants are defined in section 4.1.2.2.

Format:

```
function processKey(key)
{
   /* processing code */
}

<MenuOrForm_name>.onKey = processKey;
```

Table of Contents

4.2 reader

The `reader` object models the Code Reader hardware and firmware. Use the methods and properties of the reader object to command the behavior of the Code Reader such as:

- Executing commands on the Code Reader
- Running a JavaScript on the Code Reader
- Reading and changing Code Reader settings
- Obtaining data decoded from bar codes

Table of Contents

4.2.1 Methods

This section documents the methods defined for the Code Reader's `reader` object.

Table of Contents

4.2.1.1 beep

The `beep` method causes the Code Reader to beep.

Format:

```
reader.beep(numBeeps);
```

Where:

`numBeeps` – number; number of beeps.

Note: This method does not return a value.

Example:

```
reader.beep(3);
```

Cause the reader to beep 3 times

Table of Contents

4.2.1.2 defaultSettings

The `defaultSettings` method resets selected Code Reader settings to manufacturing defaults; it is equivalent to sending the 'J' command using the `reader.processCommand` method (section 4.2.1.3).

Format:

```
reader.defaultSettings();
```

Note: This method has no arguments and no return value.

Code Reader settings are defined in *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450, which also identifies settings that this command does not reset.

Table of Contents

4.2.1.3 processCommand

The `processCommand` method instructs the Code Reader to execute a command.

Format:

```
result = reader.processCommand(commandType, data);
```

Where:

`commandType` – string, 1 character; the command to be processed on the Code Reader.

`data` – string; data as required to process the command.

`result` – depending on the command, either:

- a Boolean value
- a data string

For `commandType`, `data`, and resulting values, see *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

Example:

```
reader.processCommand('$', "\x03"); // read a code
```

Sends a “$” command code (post event) with a one-byte value of 3 (event type = read near and far fields) to the Code Reader firmware.

Table of Contents

4.2.1.4 readSetting

The `readSetting` method returns the current value of the specified configuration setting.

Format:

```
value = reader.readSetting(settingNumber);
```

Where:

`settingNumber` – number; integer value representing the setting to be read.

For `settingNumber` values, see *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

Example:

```
value = reader.readSetting(0x1b);
```

Returns the current value of the Code Reader setting hex 1b (communications mode).

Table of Contents

4.2.1.5 runScript

The `runScript` method instructs the Code Reader to schedule the load, compile, and execution of the specified JavaScript. The Code Reader schedules execution of the script immediately after the currently executing event handler or main script completes. The `runScript` method does not include a mechanism to return to the calling script.

Format:

```
result = reader.runScript(scriptName);
```

Where:

`scriptName` – string; the name of the JavaScript to be run. The script must first be loaded into Code Reader flash by name. See the Download Utility (section 1.5).

`result` – Boolean; `true` if the script was loaded successfully; `false` otherwise. A return of `false` usually means that the script could not be found.

Example:

In the forms example (section 3.2.2), the onTimeCard function could be defined as follows:

```
function onTimeCard()
    {reader.runScript("TimeCardApp.js");}
```

The operator, at the end of a work shift, could press the "TimeCard" button to access a time card application.

Table of Contents

4.2.1.6 saveSettings

The `saveSettings` method writes the current values of the Code Reader configuration settings into flash memory. Operational setting values are loaded from flash memory when the Code Reader initializes. Any changed configuration settings will be lost at reader shutdown unless saved in flash memory.

Format:

```
result = reader.saveSettings();
```

Where:

`result` – Boolean; `false` if the flash write fails; `true` otherwise.

Note: There are no arguments to this method.

Table of Contents

4.2.1.7 setDisplayLed

The setDisplayLed method activates the LED of the CR3 above the display.

Format:

```
reader.setDisplayLed(color);
```

Where:

color – must be reader.green, reader.red, reader.amber, or reader.none.

Note: Setting 0x014d must be set to false for setDisplayLed to function properly. Requires firmware 3478+.

Table of Contents

4.2.1.8 setInterval

The setInterval method works similarly to the HTML DOM setInterval method, except that the resolution is in seconds rather than milliseconds.

Format:

```
intervalId = reader.setInterval(function, interval_sec);
```

Where:

intervalId – program provided interval ID.

function – program provided function to run at the specified interval.

interval_sec – amount of time (in seconds) to delay before running the function again.

Note: Requires firmware 3478+.

Table of Contents

4.2.1.9 clearInterval

The clearInterval method works similarly to the HTML DOM clearInterval method, and is used in conjunction with setInterval to stop processing a function called by setInterval.

Format:

```
reader.clearInterval(intervalId);
```

Where:

intervalId – program provided interval ID.

Note: Requires firmware 3478+.

Table of Contents

4.2.1.10 setTimeout

The `setTimeout` method works similarly to the HTML DOM setTimeout method, except that the resolution is in seconds rather than milliseconds.

Format:

```
timeoutId = reader.setTimeout(function, timeout_sec);
```

Where:

`timeoutId` – program provided timeout ID.

`function` – program provided function to run after the specified timeout.

`timeout_sec` – amount of time (in seconds) to delay before running the function.

Note: Requires firmware 3478+.

Table of Contents

4.2.1.11 clearTimeout

The `clearTimeout` method works similarly to the HTML DOM clearTimeout method, and is used in conjunction with setTimeout to stop processing a function called by setTimeout.

Format:

```
reader.clearTimeout(timeoutId);
```

Where:

`timeoutId` – program provided timeout ID.

Note: Requires firmware 3478+.

Table of Contents

4.2.1.12 shiftJisToUnicode

The `shiftJisToUnicode` method converts a string from Shift-JIS encoding to Unicode encoding.

Format:

```
unicodeString = reader.shiftJisToUnicode(text);
```

Where:

`text` – String; text encoded as JIS.

`unicodeString` – String; text encoded as Unicode.

Example:

```
myUnicodeString = reader.shiftJisToUnicode(myString);
```

Sets myUnicodeString to the Unicode encoded equivalent of myString.

Table of Contents

4.2.1.13 writeSetting

The `writeSetting` method changes the operational value of a single Code Reader configuration setting.

Format:

```
writeSetting(settingNumber, value);
```

Where:

`settingNumber` – number; the setting to be changed.

`value` – number; the value to be written to the configuration setting.

For the possible values of `settingNumber` and `value`, see *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

Note: This method does not return a value.

Example:

```
reader.writeSetting(0x1b, 4);
```

Sets the reader communications mode to Bluetooth RF. See also the `gui.Button` example in section 4.1.3.1.

Table of Contents

4.2.2 Properties

This section documents the properties defined for the Code Reader's `reader` object.

Table of Contents

4.2.2.1 onBatteryLevelChange

The `onBatteryLevelChange` property of the `reader` object provides processing control when the Code Reader detects a change in its battery charge level.

Format:

```
function batteryCharge( previousLevel,
                        currentLevel )
{
   /* Processing statements */
}
reader.onBatteryLevelChange = batteryCharge;
```

Where:

`previousLevel` – integer; previous battery charge level.

`currentLevel` – integer; current battery charge level.

Possible battery charge levels are documented in sections 4.2.2.9 through 4.2.2.12.

Example:

```
function batteryCharge( previousLevel,
                        currentLevel )
{
   if ( currentLevel == reader.amber )
        alert("Battery Low");
}
reader.onBatteryLevelChange = batteryCharge;
```

Sends an alert when the battery level drops to amber.

Table of Contents

4.2.2.2 onCommand

The `onCommand` property of the Code Reader calls the specified function when the reader:

- Receives a configuration command from a communication port.
- Decodes a configuration command from a code read by the Code Reader.

The application uses this property as an event handler to:

- Receive notification of command processing.
- Prevent execution of a command.

The function will not be called in response to a `reader.processCommand` call or commands within a stored-code (“performance strings”). Performance strings are documented in *Code Reader 3 – Reader-Host Interface Specification* Code Document Number V002450.

Return Boolean `true` to instruct the reader to process the command. Return Boolean `false` to suppress the command. When a command is suppressed, the firmware will not send any response to the host, but the JavaScript application may provide its own response to the host.

Format:

```
function filterCommand(commandType, commandData)
{
   var shouldSuppressCommand = false;

   /* Processing statements */

   return !shouldSuppressCommand;
}
reader.onCommand = filterCommand;
```

Where:

`commandType` – string; 1 character; specifies the command being processed.

`commandData` – string; data to be process by the command.

Example:

```
function notifyErase(commandType)
{
   if ( commandType == ')' )
         print("Erasing Error Log...");
}
reader.onCommand = notifyErase;
```

Sends a debugging message to the host to show that the erase command was detected.

Table of Contents

4.2.2.3 onCommandFinish

The `onCommandFinish` property of the `reader` object provides processing control upon completion of a command.

Format:

```
function finishedCommand(commandSuccess,
                         responseType,
                         responseData)
{
   /* Processing statements */
}
reader.onCommandFinish = finishedCommand;
```

Where:

`commandSuccess` – Boolean; contains the return status of the command: true = success, false = failure.

`responseType` – string; 1 character; specifies the response type.

`responseData` – string; the response data.

Example:

```
function finishedCommand(commandSuccess,
                         responseType,
                         responseData)
{
   if( !commandSuccess )
        alert("Command failed ("
              + responseType + ":" + responseData + ")");
}
reader.onCommandFinish = finishedCommand;
```

sends an alert when a command fails.

Table of Contents

4.2.2.4 onDecode

The `onDecode` property of the `reader` object provides processing control to the application program at the completion of a decode action. The Code Reader firmware passes the decode object to the function through the calling argument.

Code the function in your script and return a code as follows:

`null` – the decode has been consumed by the JavaScript application; there should be no further processing of it by the Code Reader firmware.

`false` – invalidate the decode; if the Code Reader firmware is so-configured, it will act as if there had not been a decode; the good-decode-beep will be suppressed.

`decode` object (modified or unmodified) – the Code Reader firmware will continue to process the modified or unmodified decode data.

Format:

```
function onDecode(decode)
{
   var valid       = true;
       /* set to false below if decode is to be invalidated */

   var passthrough = true;
       /* set to false below if decode is consumed here */

   /* processing statements, which may modify decode.data,
      valid, and/or passthrough */
```

```
        if( !valid )
            return false;

        if( !passthrough )
            return null;

        return decode;
    }
    reader.onDecode = onDecode;
```

Where:

decode – object having the following properties:

data – string; the text decoded from the bar code.

symbology – read-only number; the symbology number (see *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450).

symbologyModifier – read-only number; the symbology modifier number (see *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450).

symbologyIdentifier – read-only string; this is the AIM identifier ("]cm").

x – read-only number; unit is pixels, 0 is center of image.

y – read-only number; unit is pixels, 0 is center of image.

x,y combined specify the position of the center of the bar code in the image (relative to the center of the image; the values can be positive or negative).

time – read-only Date object; a JavaScript Date object indicating the time the code was read.

quality_percent – read-only number; a code quality metric returned by the decoder. The precise meaning is symbology-specific.

linkage – read-only number; indicates that a code is one part of a composite code. See *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

bounds – 4-element array, indexed from 0 – 3. Each element is a decode.bounds object with 2 properties: x and y, both are integers and read only. Note: Requires firmware 3280+.

Example:

See the discussion of symbol decoding in section 3.4.

Table of Contents

4.2.2.5 onDecodeAttempt

The `onDecodeAttempt` property of the `reader` object provides processing control to the application program at the completion of a decode action, before any of the decoded symbols are passed to `reader.onDecode`.

Format:

```
function onDecodeAttempt(count)
{
   /* processing statements */
}
reader.onDecodeAttempt = onDecodeAttempt;
```

Where:

`count` – number; a count of the number of symbols that were read by a single decode request.

Note: This method does not return a value.

Example:

```
var ok = false;

reader.onDecodeAttempt = function(count)
{
    ok = count >= 2;
}

reader.onDecode = function(decode)
{
    if( !ok )
        return false;

    return decode;
}
```

Ensures there at least two decodes per attempt; otherwise, invalidates the single decode. Each decode found in the field of view will be decoded only once per attempt, so this example ensures there are two distinct symbols in the field of view. The reader must have been configured (section 3.7) to support multiple reads per attempt.

Table of Contents

4.2.2.6 onIdle

The `onIdle` property of the `reader` object provides processing control to the application program whenever the reader is idle; i.e., no events (such as button presses) are active or queued.

This event is posted when the JavaScript has nothing else queued and is not related to the Code Reader active time (setting hex 32).

Format:

```
function onIdle()
{
   /* processing statements */
}
reader.onIdle = onIdle;
```

Note: This method does not return a value.

Example:

```
function onIdle()
{
    reader.processCommand('.', "\x22\x05\x32\x64");
}

reader.onIdle = onIdle();
```

Flashes both LEDs on the CR2 green 5 times, with LEDs on for ½ second and off for 1 second.

Note: Requires firmware 3280+

Table of Contents

4.2.2.7 onStandby

The `onStandby` property of the `reader` object provides processing control to the application program whenever the reader is is about to enter the standby mode.

Format:

```
function onStandby()
{
   /* processing statements */
}
reader.onStandby = onStandby;
```

Where:

`return` – Boolean; true if the reader should be allowed to enter the standby mode; false to prevent it.

Example:

```
function onStandby()
{
    if (comm.isConnected) return false;
    else return true;
}
```

```
reader.onStandby = onStandby();
```

Prevents the reader from entering standby if it is connected and allows it to enter standby otherwise.

Table of Contents

4.2.2.8 batteryLevel

The `batteryStatus` property of the `reader` object contains a read only integer specifying the battery charge level. Possible battery charge levels are:

`reader.green` – not low.

`reader.amber` – somewhat low.

`reader.red` – very low.

`reader.none` – battery not present.

Example:

```
batteryLevel = reader.batteryLevel;
```

Table of Contents

4.2.2.9 red

The `red` property of the `reader` object contains a read only constant for use with reader.batteryLevel and reader.setDisplayLed.

Note: Requires firmware 3478+

Table of Contents

4.2.2.10 green

The `green` property of the `reader` object contains a read only constant for use with reader.batteryLevel and reader.setDisplayLed.

Note: Requires firmware 3478+

Table of Contents

4.2.2.11 amber

The `amber` property of the `reader` object contains a read only constant for use with reader.batteryLevel and reader.setDisplayLed.

Note: Requires firmware 3478+

Table of Contents

4.2.2.12 none

The `red` property of the `reader` object contains a read only constant for use with reader.batteryLevel and reader.setDisplayLed.

Note: Requires firmware 3478+

Table of Contents

4.2.2.13 cabled

The `cabled` property of the `reader` object contains a read only Boolean value containing the cabling state of the Code Reader hardware. The value will be `true` if cabled and `false` if not cabled.

Example:

```
   cabled = reader.cabled;
```

Note: Requires firmware 3478+.

Table of Contents

4.2.2.14 charging

The `charging` property of the `reader` object contains a read only Boolean value containing the charging state of the Code Reader hardware. The value will be `true` of charging and `false` if not charging.

Example:

```
   charging = reader.charging;
```

Note: Requires firmware 3280+.

Table of Contents

4.2.2.15 hardwareVersion

The `hardwareVersion` property of the `reader` object contains a read only string containing the version number of the Code Reader hardware.

Example:

```
   hwVersion = reader.hardwareVersion;
```

Table of Contents

4.2.2.16 oemId

The `oemId` property of the `reader` object contains a read-only string containing the Code Reader unique OEM identifier from the locked flash memory.

Example:

```
   oemId = reader.oemId;
```

Table of Contents

4.2.2.17 readerId

The `readerId` property of the `reader` object contains a read-only string containing the Code Reader unique ID from the locked flash memory.

Example:

```
   rid = reader.readerId;
```

Table of Contents

4.2.2.18 softwareVersion

The `softwareVersion` property of the `reader` object contains a read only string containing the version number of the firmware currently running in the Code Reader.

Example:

```
   swVersion = reader.softwareVersion;
```

Table of Contents

4.3 storage

The `storage` object provides application software access to Code Reader file storage. Files are written to storage by the `storage.write` method and by downloading from the host (see section 3.6).

Note: Names of files can be 1 - 200 printable ASCII characters. For compatibility with host file systems, Code recommends you do not use characters that are reserved by host operating systems: /, \, :, ?, *, [,], ', ", etc. Files should be kept to a maximum length of 32K bytes. Files are stored in UTF8 format, which encodes Unicode characters in one or more bytes each.

Table of Contents

4.3.1 Methods

The following section documents the methods defined for the Code Reader `storage` object.

In this section, the examples use elements of a time card application that assumes time card records are maintained as files organized by employee number. The naming convention for the time card records is `TimeCard<employee_number>.`

Table of Contents

4.3.1.1 append

The `storage.append` method adds data to the end of a file.

Format:

```
result = storage.append(name, data);
```

Where:

`name` – string; the name of the object to append.

`data` – string; the data to add to the end of the file.

`result` – Boolean; `true` if the append succeeded; `false` if the append failed.

Example:

```
storage.append("TimeCard” + employeeNumber, tcRecord);
```

Adds the time card record to the end of the time card record that already exists for the employee specified by `employeeNumber`.

Note: Requires firmware 3226+

Table of Contents

4.3.1.2 erase

The `storage.erase` method erases a file.

Format:

```
result = storage.erase(name);
```

Where:

`name` – string; the name of the object to erase.

`result` – Boolean; `true` if the file existed (the object is deleted); `false` if the file did not exist.

Example:

```
storage.erase("TimeCard" + employeeNumber);
```

Erases the time card record for the employee specified by `employeeNumber`.

Table of Contents

4.3.1.3 findFirst

The `storage.findFirst` method locates the first file where the name matches a regular expression specified in the call parameter.

Format:

```
name = storage.findFirst(expression);
```

Where:

`expression` – regular expression (not a string); a regular expression used by the Code Reader to match against names of stored objects.

`name` – string; the name of the first matching file; `name` is `null` if no file matches the `expression`.

Example:

```
name = storage.findFirst(/^TimeCard.*/);
```

Sets `name` to the name of the first time card record file.

Table of Contents

4.3.1.4 findNext

The `storage.findNext` method locates the next file where the name matches the regular expression specified in the `expression` parameter of a previous `storage.findFirst` call.

The matching names are not ordered, but they will not be repeated; a `findFirst - findNext` sequence will return all matching files, provided that there are no other intervening storage method calls. (You can put the files into an array and use JavaScript's sort method when you need them ordered.)

Format:

```
name = storage.findNext();
```

Where:

`name` – string; the name of a file; `name` is `null` if no remaining file matches the previous regular expression.

Example:

```
name = storage.findNext();
```

Sets `name` to the name of the next time card record file.

Table of Contents

4.3.1.5 read

The `storage.read` method reads a file.

Format:

```
data = storage.read(name);
```

Where:

`name` – string; the `name` of a file.

`data` – string; the contents of the file; `null` if there was no file with that name.

Example:

```
data = storage.read(name);
```

Sets `data` to the contents of the time card record specified by `name`.

Table of Contents

4.3.1.6 size

The `storage.size` method returns the size of a file in bytes.

Format:

```
nameSize = storage.size(name);
```

Where:

`name` – string; the `name` of a file.

`nameSize` – integer; the size of the file in bytes.

Example:

```
nameSize = storage.size(“name”);
```

Sets `nameSize` to the size of the time card record specified by `name`.

Note: Requires firmware 3280+

Table of Contents

4.3.1.7 upload

The `storage.upload` method uploads a file to the host over the current active host comm port.

Format:

```
result = storage.upload(name, withHeaderAndFooter);
```

Where:

`name` – string; the name of a file.

`withHeaderAndFooter` – Optional boolean; If set to false the file is uploaded without the header (ap/g(file size))and footer (ap/d(checksum)). If the parameter is not included the header and footer will be included with the upload.

`result` – Boolean; `false` if there was a failure on the communications port; otherwise, `true`. If the current communications mode is a 2-way mode, `true` indicates that the data has been sent to and acknowledged by the host.

Note: The upload protocol is documented with the "^" command in *Code Reader 3 – Reader-Host Interface Specification* Code Document Number V002450. Uploaded files may be split into multiple packets as defined in the protocol.

Example:

```
name = storage.findFirst(/TimeCard.*/);
while (name)
{
   if ( !storage.upload(name) )
         alert(name + " upload failed!");
   name = storage.findNext();
};
```

Uploads all time card records to the host. If a time card record fails to upload, the operator is alerted.

Table of Contents

4.3.1.8 write

The `storage.write` method writes a file to storage. If the file does not exist, the Code Reader creates it. If there was an existing file of the same name, it is replaced.

Format:

```
result = storage.write(name, data);
```

Where:

`name` – string; name of a file.

`data` – string; data to be written.

`result` – Boolean; `true` if the file was successfully written; otherwise, `false`.

Note: When replacing an existing file, if there is insufficient storage space to hold the new file, it will not be written; however, the old file will be erased.

Example:

```
result = storage.write("TimeCard" + employeeNumber, tcRecord);
```

Writes a time card record to a file.

Table of Contents

4.3.2 Properties

The following section documents the properties defined for the Code Reader `storage` object.

Table of Contents

4.3.2.1 fullness_percent

The `storage.fullness_percent` property is a read-only integer containing the percent of storage in use.

Table of Contents

4.3.2.2 isFull

The `storage.isFull` property is a read-only Boolean value; `true` if storage is full and cannot be added to; otherwise, `false`.

Table of Contents

4.4 comm

The comm object models the host commutation feature of the Code Reader. Use the methods and properties of the comm object to send either packet or text data to the host.

Table of Contents

4.4.1 Methods

The following section documents the methods defined for the Code Reader comm object.

Table of Contents

4.4.1.1 connect

The connect method instructs the Code Reader communication driver to attempt to establish a connection.

Format:

```
result = comm.connect(timeout_sec);
```

Where:

timeout_sec – integer, the number of seconds for the communication driver to continue to attempt to establish a connection.

result – Boolean; false if there was a failure to connect; otherwise, true.

Example:

```
result = comm.connect(30);
```

Causes the reader to attempt to connect for up to thirty seconds. The reader stops attempting to connect when either a connection is made or the timeout is reached (i.e. if a connection is established after three seconds, the reader does not wait for the remaining twenty seven seconds before moving to the next queued task).

Note: Requires firmware 3280+

Table of Contents

4.4.1.2 disconnect

The disconnect method instructs the Code Reader communication driver to disconnect from the host.

Format:

```
comm.disconnect();
```

Example:

```
comm.disconnect();
```

Causes the reader to disconnect from the host.

Note: This method does not return a result. Requires firmware 3280+

Table of Contents

4.4.1.3 sendPacket

The `sendPacket` method instructs the Code Reader to send a data packet to the host via the communications port currently specified by the active Code Reader communication settings. The Code Reader creates a packet formatted according to the active Code Reader packet protocol configuration setting.

For a discussion of data packets, see *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

Format:

```
result = comm.sendPacket(type, data);
```

Where:

`type` – string, length 1; the type of packet to send. The packet types are documented in *Code Reader 3 – Reader-Host Interface Specification*, Code Document Number V002450.

`data` – string; data to be inserted into the packet.

`result` – Boolean; `false` if there was a failure on the communications port; otherwise, `true`. If the current communications mode is a 2-way mode, `true` indicates that the data has been sent to and acknowledged by the host.

Example:

```
reader.onDecode =
   function(decode) {comm.sendPacket('z', decode.data)};
```

Sends a packet containing results of a decode to the current comm port.

Table of Contents

4.4.1.4 sendText

The `sendText` method instructs the Code Reader to send arbitrary text (which may include NULL characters) to be sent via the active communication port; the text will be sent “raw” regardless of the reader comm mode settings. This method buffers the data until the USB packet size limit is reached or a 'z' packet is sent. For an immediate response, send the data as a 'z' packet using comm.sendPacket.

Format:

```
result = comm.sendText(data);
```

Where:

`data` – string; data to be sent via the active communication port.

`result` – Boolean; `false` if there was a failure on the communications port; otherwise, `true`. If the current communications mode is a 2-way mode, `true` indicates that the data has been sent to and acknowledged by the host.

Example:

```
reader.onDecode =
function(decode) {comm.sendText("decode.data"); }
```

Sends the raw text "decode.data" via the active communications port.

Note: Requires firmware 3280+

Table of Contents

4.4.2 Properties

The following section documents the properties defined for the Code Reader `comm` object.

Table of Contents

4.4.2.1 isConnected

The `isConnected` property of the `comm` object contains a read-only Boolean specifying the host connection status. Possible connection values are:

`true` – reader is connected to the host.

`false` – reader is not connected to the host.

Example:

```
connected = comm.isConnected;
```

Table of Contents

4.5 Functions

The following section documents functions that enhance the application development environment.

Table of Contents

4.5.1 Dialog

The Code Reader JavaScript Engine provides the following functions like those defined by JavaScript in Web browsers:

- alert
- confirm
- prompt

These functions interact with the CR3 standard GUI display. The CR3 displays the name of the function in the GUI status bar and the text associated with the function, and then waits until a key is pressed. The following subsections describe the operation of each function in the CR3 environment.

Similar but more flexible functions are provided in the `gui` object (see section 4.1). For example, if you want to change the caption on these displays use the `gui` object functions.

Table of Contents

4.5.1.1 alert

The `alert` function displays text in the display area of the standard GUI display. Do not call this function within onDecode and onCommand event handlers.

Format:

```
alert(text);
```

Where:

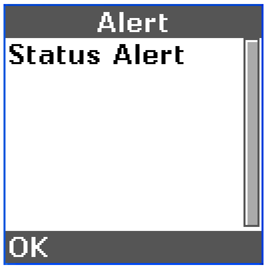
`text` – string; text to display as the alert.

Processing suspends until the operator presses an enter key – either the enter key or the left softkey defined as OK.

Example:

```
alert("Status Alert");
```

Displays the alert shown in Figure 17 and waits until the operator presses the enter key or the left softkey (OK).



Status Alert

Figure 17 – Alert Example

Table of Contents

4.5.1.2 confirm

The `confirm` function displays text in the display area of the standard GUI display and returns a value based on the key pressed. Do not call this function within onDecode and onCommand event handlers.

Format:

```
result = confirm(text);
```

Where:

`text` – string; text to display for confirmation.

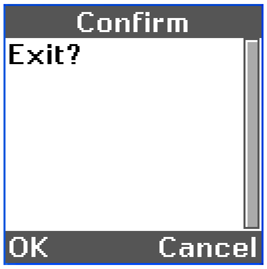
`result` – Boolean; `true` if the confirm receives an enter key (either the enter key or the left softkey defined as `OK)`; `false` if the confirm receives the right softkey defined as `Cancel`.

Processing suspends until the operator presses a suitable key.

Example:

```
result = confirm("Exit?");
```

Displays the confirm shown in Figure 18 and waits until the operator presses the left softkey (OK) or the right softkey (Cancel).



Exit?

Figure 18 – Confirm Example

If you want softkey labels other than OK and Cancel (for example, Yes and No), use the `gui.confirm` method (section 4.1.1.2).

Table of Contents

4.5.1.3 prompt

The `prompt` function displays text in the display area of the standard GUI display and returns a value based on the key pressed. Do not call this function within onDecode and onCommand event handlers.

Format:

```
result = prompt(text, default);
```

Where:

`text` – string; text to display as a label above a `gui.Edit` control.

`default` – string; a default string to display as the contents of edit control.

`result` – string; contents of the edit control if the prompt receives an enter key (either the enter key or the left softkey defined as `OK)`; null if the prompt receives the right softkey defined as `Cancel`.

Processing suspends until the operator presses an enter key or `Cancel` key. The operator can key new data into the edit control before pressing enter or the left softkey.

Example:

```
string = prompt("Enter login ID", "None");
```

Displays the prompt shown in Figure 19.

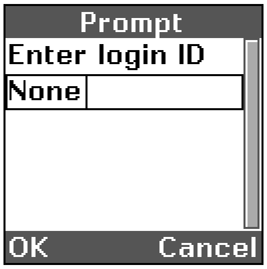


Figure 19 – Prompt Example

The value of `string` depends on the operator action.

- If the operator at any time presses the right softkey (Cancel), the value of `string` is null.
- If the operator changes the contents of the edit control to <new content> and presses the left softkey (OK), the value of `string` is <new content>.

- If the operator presses the left softkey (OK) without changing the contents of the edit control, the value of `string` is “None” (the value entered as the second call parameter).

Table of Contents

4.5.2 Process Control

4.5.2.1 sleep_ms

The Code Reader defines a sleep function to control time-sequence. Any busy-loops or time-consuming tasks should sleep to give other tasks time to run. It is very important to include a sleep_ms(0) function in your code periodically to give the main task a chance to update the watchdog timer. Failure to do so will cause a watchdog timeout error.

Event handlers such as `onClick` run during main code sleep. The event handlers themselves must NOT sleep; they should handle the event and return as quickly as possible.

Format:

```
sleep_ms(milliseconds);
```

Where:

`milliseconds` – number; the minimum number of milliseconds to sleep.

Table of Contents

4.5.3 Other Functions

4.5.3.1 format

The `format` function allows you to combine variables and text into a string. Its operation is similar to the `sprintf` function of the C language.

Format:

```
string = format(<control_string>, <argument_list>);
```

Where:

`<control_string>` – contains a combination of characters that will be included in the string and format specifiers that instruct `format` how to process the items in the argument list.

<argument_list> – a comma-separated list of items to be processed according to format specifiers in the control string.

Example:

```
n = 45;
s = "ID";
string = format("%s = %d", s, n);
```

creates the string:

```
"ID = 45"
```

Format specifiers are taken from the standard C library and are discussed in Appendix C.

The output string is truncated to 1023 characters. If an error occurs, the output string is “format error.”

Table of Contents

4.5.3.2 include

The `include` function executes the included script inline.

Format:

```
result = include(scriptName);
```

Where:

`scriptName` – string; the name of the script to be included.

`result` – Boolean; `true` if the script could be loaded and executed; otherwise, `false`.

Example:

```
include("myScript.js");
```

adds the definitions in `myScript.js` to the application. The definitions become part of the “including” script.

Table of Contents

4.5.3.3 print

The `print` function sends text to stdout (the active communication port), not to the CR3 display. Limit the use of the `print` function to debugging. Use the `comm` object methods for normal data output to communication ports.

Format:

```
print(text);
```

Where:

`text` – string; debugging data to be sent to the active communications port.

Table of Contents

4.5.3.4 setStandbyMessage

The `setStandbyMessage` allows you to create a custom standby message to display when the reader enters standby mode.

Format:

```
setStandbyMessage(text);
```

Where:

`text` – string; message to display when the reader enters standby mode.

Note: Requires firmware 3280+

Table of Contents

Glossary and Acronyms

Term	Definition
Control	User Class object instantiated in a CR3 GUI form.
CR3	Code Corporation Code Reader 3
RF	Radio Frequency
Code Data	Data resulting from the decode process after data capture or bar code read
Smart Quote	Previously formatted quotation marks, usually found in a word processing program
Softkey	User programmable key found on the CR3
Consume	Used with no return value by the user defined application or firmware

Table of Contents

Appendix A Code Reader 3 Simulator

Code provides a JavaScript simulator as part of the CR3 Application Development environment. A free source code editor, `SciTE`, is packaged with the simulator.

From the editor you can execute the current edit file and walk through JavaScript errors detected during execution.

Table of Contents

A.1 Installation

The simulator/editor package is distributed as a `.zip` file. To install, simply unzip the file into any directory in your Microsoft® Windows® environment. This document refers to this installation directory as the base directory. The unzip process creates two subdirectories, `editor` and `jse`, and a shortcut, `JSE.exe`, to the `SciTE` editor tailored to the CR3 simulator.

The editor directory contains the editor and associated operational files, and SciTE documentation. The file `editor/SciTEDoc.html` contains the editor user manual. The directory contains additional SciTE html documents that discuss an array of extensions, add-ons, and programming interfaces. These discussions are beyond the scope of this document.

The `jse` directory contains the CR3 JavaScript simulator and associated operational files. When you start the `JSE.exe` program, the directory `jse` becomes the default directory for script files.

Table of Contents

A.2 Using JSE

To execute the editor, double click on the JavaScript icon in the base directory.

JSE

JSE displays an editor window. From there, you can run the simulator (section A.2.2).

Table of Contents

A.2.1 Editor Window

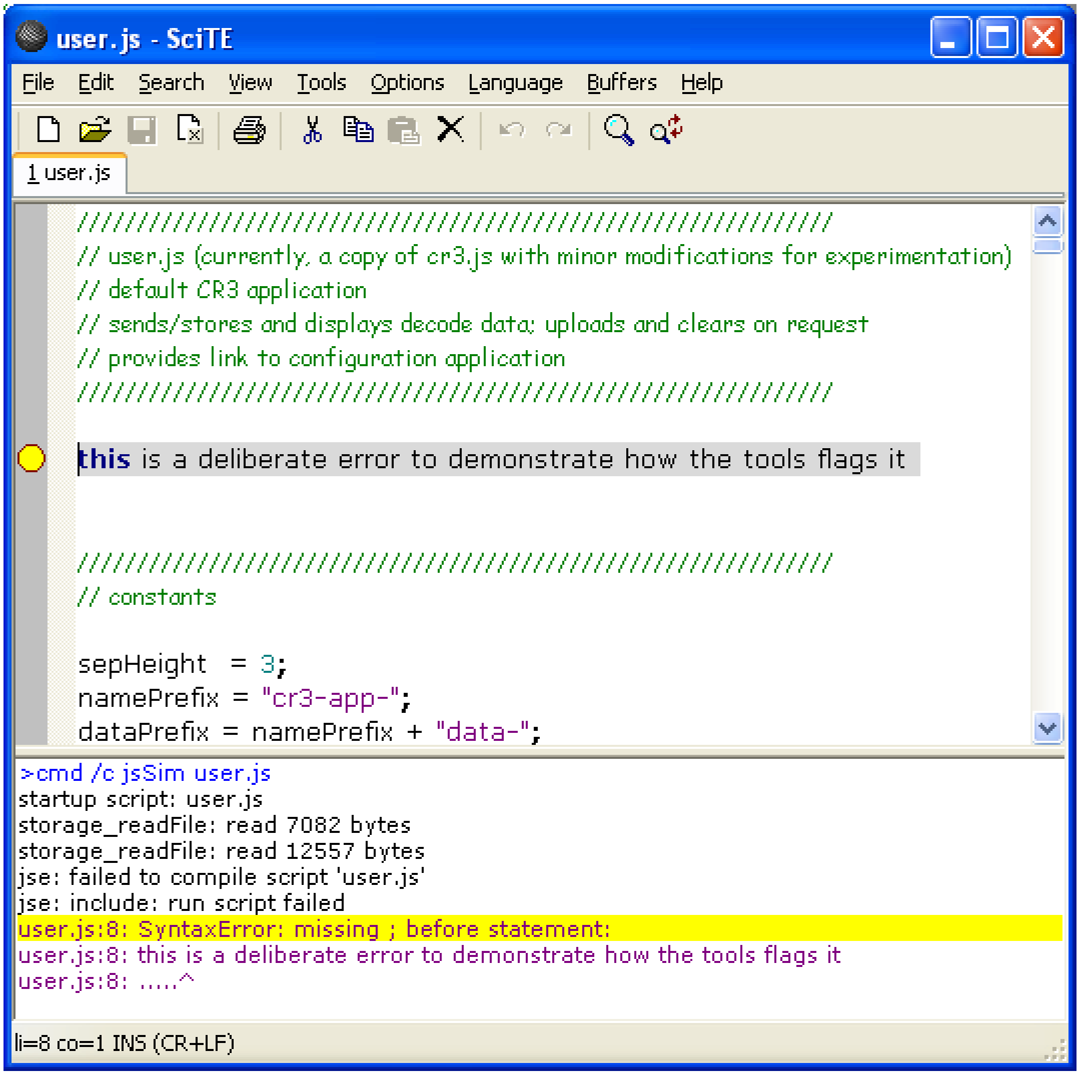
The editor displays the window shown in Figure 20, which shows the execution of a script, `user.js`, which purposely includes an error to demonstrate the editor display.

Two keys control execution and error evaluation when the editor window has focus: function key 4 (`F4`) and function key 5 (`F5`).

- `F4` steps through detected errors when repeatedly pressed.
- `F5` instructs the editor to execute the currently selected script.

For additional controls and features of the editor, see the `SciTE` user documentation in `<base_directory>/editor/SciTEDoc.html`.

In Figure 20 the `F5` key has been pressed to start execution of the script, and the `F4` key has been pressed to highlight the first error. Note the yellow circle at the left of the display that highlights the currently selected statement in error. Note: SciTE includes an option to display line numbers (see the SciTE `View` menu).



```
////////////////////////////////////////////////////////////////
// user.js (currently, a copy of cr3.js with minor modifications for experimentation)
// default CR3 application
// sends/stores and displays decode data; uploads and clears on request
// provides link to configuration application
////////////////////////////////////////////////////////////////

this is a deliberate error to demonstrate how the tools flags it


////////////////////////////////////////////////////////////////
// constants

sepHeight  = 3;
namePrefix = "cr3-app-";
dataPrefix = namePrefix + "data-";
```

```
>cmd /c jsSim user.js
startup script: user.js
storage_readFile: read 7082 bytes
storage_readFile: read 12557 bytes
jse: failed to compile script 'user.js'
jse: include: run script failed
user.js:8: SyntaxError: missing ; before statement:
user.js:8: this is a deliberate error to demonstrate how the tools flags it
user.js:8: .....^
```

Figure 20 – Editor Display

Table of Contents

A.2.2 Simulator Window

Figure 21 shows both segments of the CR3 simulator window. The upper segment, `CR3 Simulator`, simulates the display screen on the CR3. The lower segment, `Simulated Decode`, contains a data entry control into which you can type text to simulate scanning a bar code (key in or copy and paste data and press enter). It may be necessary to input characters that cannot be keyed in. To input these characters, use URL encoding (% followed by the hexadecimal value of the character). For example, <SOH>1234<EOT> would be encoded as %011234%04. The simulated decode window can be resized, but does not support multiple line input.

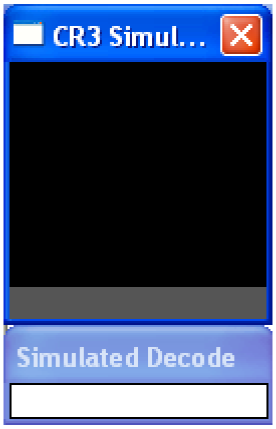


Figure 21 – CR3 Simulator Display

The standard computer keyboard mappings simulate the keypad of the CR3 as follows:

- F1 simulates the left CR3 softkey.
- F2 simulates the right CR3 softkey.
- Backspace simulates the CR3 clear key.
- Enter simulates the blue key in the CR3 cursor pad.
- The arrow, shift, and number keys simulate the corresponding CR3 keys.
- Alt+F4, or typing “q” twice, closes both segments of the CR3 Simulator Display. (You can also close the display by clicking the `CR3 Simulator` close (“X”) button.)

For a complete discussion of the CR3 key pad, see the *Code Reader 3 – User Manual*, Code Document Number C001537.

Table of Contents

Appendix B Input Modes

The input mode determines the character set that is active for the CR3 keypad. The modes are described in Table 2.

Table 2 – Keypad Input Modes

<table>
<tr><th>inputMode</th><th>characters</th></tr>
<tr><td>numeric</td><td>0123456789</td></tr>
<tr><td>caps</td><td>A-Z, 0-9 and all ASCII non-alphanumeric symbols:
'!', '"', '#', '$', '%', '&', '\'', '(',
')', '*', '+', ',', '-', '.', '/', ':', ';',
'<', '=', '>', '?', '@', '[', '\\', ']', '^',
'_', '`', '{', '|', '}', '~'</td></tr>
<tr><td>lower</td><td>a-z, 0-9 and all ASCII non-alphanumeric symbols</td></tr>
<tr><td>latinCaps</td><td>All characters in caps plus all accented capital letters from the ISO-8859-1 character set and the additional ISO-8859-1 non-alphanumeric symbols</td></tr>
<tr><td>latinLowerCase</td><td>All characters in lowercase plus all accented lowercase letters from the ISO-8859-1 character set and the additional ISO-8859-1 non-alphanumeric symbols</td></tr>
<tr><td>symbols</td><td>All ASCII and ISO-8859-1 non-alphanumeric symbols</td></tr>
</table>

Table of Contents

Appendix C Format Specifiers

The control string of the format function accepts the following codes from the standard C library:

%d	signed decimal integers
%i	signed decimal integers
%e	lowercase scientific notation
%E	uppercase scientific notation
%f	floating point decimal
%g	uses %e or %f , whichever is shorter
%G	uses %E or %f, whichever is shorter
%o	unsigned octal
%s	character string
%u	unsigned decimal integers
%x	lowercase unsigned hexadecimal
%X	uppercase unsigned hexadecimal
%%	insert a percent sign

Flag, width, and precision modifiers are the same as in the standard C library definition.

Table of Contents