pronto <script>

EN Developer's Guide

pronto

PHILIPS

ProntoScript Developer's Guide

Version 1.2.3
2009-07-22T13:17:48+02:00

Abstract

The next generation of the Pronto Professional control panels provide the ability to use `ProntoScript` scripts, based on the popular `JavaScript` scripting language.

This developer guide describes the features ProntoScript provides, and can be used when developing scripted Control Panel configurations.

Conditions to the use of the Pronto Script Developers Guide

1. Pronto Script Developers Guide and its documentation (hereafter *"PSDG"*)

   The PSDG has been written by Philips for owners and users of Pronto products as guidance to develop new software-modules in Pronto Script. The PSDG is destined to be used only by persons, who are professional users or installers of Pronto products and who are trained to use the PSDG to develop software-modules in Pronto Script (hereafter *"User"*).

2. Intellectual property and ownership

   The PSDG is intellectual property of Philips. By using the PSDG User agrees that the PSDG is, shall be and shall remain the intellectual property of Philips. User shall immediately cease using the PSDG upon first demand of Philips.

3. License grant

   Under its intellectual property Philips herby grants to User a royalty-free, non-exclusive, non-transferable license solely to use the PSDG as guidance to develop software-modules in Pronto Script. The rights of User are limited to the foregoing. By using the PSDG User accepts the preceding license grant and acknowledges that the PSDG constitutes a valuable asset of Philips. Accordingly, except as expressly permitted under the license grant, User agrees not to: otherwise use the Philips Intellectual Properties; modify, adapt, alter, translate, disassemble, re-create, copy, decompile, reverse engineer, or create derivative works from the PSDG, Pronto products and/or the Pronto Script, or; sublicense, lease, rent, or otherwise transfer the PSDG to any third party.

4. Warranty and indemnification

   Philips provides the PSDG *"as is"* as courtesy to User, *"as is"* means that Philips provides the PSDG without any warranty or support. User is allowed to use the PSDG, accordingly the license grant, at its own risk and responsibility. By using the PSDG User indemnifies Philips for all claims by any party caused by or in connection with the use of the PSDG by User. Furthermore User shall not hold Philips liable for direct, indirect, special, consequential or incidental damages, including but not limited to lost profits, business interruption, or corruption or loss of data or information, caused by or in connection with the use of the PSDG by User.

Table of Contents

List of Tables

List of Examples

Preface

1. Using this guide

The guide assumes you have some background in programming, either with languages like C, C++, Java or other languages, or with JavaScript. Even so, it is built up from easy to advanced, with plenty of examples to make the process of getting familiar with ProntoScript a fun experience:

For the experienced programmer. You can find snippets of proven, best practice code, before exploiting the full freedom of writing your own, custom code.

For the novice programmer and Pronto enthusiast. You can experiment with working, useful, real life examples that demonstrate what ProntoScript can do in automation projects.

This document does not strive for completeness. For a complete description of Javascript 1.6, on which ProntoScript is based, refer to David Flanagan's *Javascript, the Definitive Guide, 5th edition* published by O'Reilly [Flanagan].

2. What's new in 1.2

This version of this guide has been updated for ProntoEdit Professional 2.3, and the additional ProntoScript features available in the Control Panel's Platform release 2.1:

ProntoScript libraries. ProntoScript libraries can now be included in a script using the `System.include` method. (Described in Chapter 11, *Libraries*)

Widget fill color property. The fill color of a widget can now be set using the `bgcolor` property.

Controlling panel transparency. The transparency of panels which do not have a background image can be controlled using the boolean `transparent` property.

Increased dynamic widget size. The `width` and `height` properties of a `Widget` can now be set to sizes up to 65535 by 65535 pixels, where previously this was restricted to the pixel dimensions of the display.

Network interface control. The network interface can be disabled or (re-)initialized from a script. (See Section A.1.1.6, "Activity.wifiEnabled")

Power management events. On an activity level, callbacks can be defined which get called whenever the control panel enters sleep (standby) mode, or is woken up; these are the `Activity.onSleep` and `Activity.onWake` function properties of the Activity object.

Additional system information. Most of the information visible in the Info tab of the settings mode on the control panel can now be retrieved from within ProntoScript using a range of new methods of the `System` class: `getModel`, `getApplicationVersion`, `getBootloaderVersion`, `getBootloaderVersion`, `getIRVersion`, `getSerial` and `getFreeCFMemory`.

Basic popups. A new `GUI.alert` method is now available, providing an easy way for a ProntoScript programmer to show messages to the end user. (Useful for handling exceptions)

3. What's new in 1.1

The 1.1 update added added 2 major features to the ProntoScript toolset:

- Use the Rotary wheel
- Get and place an image over IP

The rotary wheel. A new activity property (see Section A.1.1.3, “Activity.onRotary”) has been defined to enable you to use the rotary wheel for any purpose. Typically it can be used for scrolling through lists showing music content.

Get and place an image over IP. This powerful function allows you to show an image (BMP, PNG or JPG: see Section A.6.1, “Image class constructor”) on the control panel that is retrieved from a source via IP. Typical applications are album art for media servers, IP cameras (still picture only; MPEG is not supported) or picture viewer. As an extra there is also a stretch property available (see Section A.13.1.9, “Widget.stretchImage”) that will allow you to even build dynamically growing, shrinking and warping buttons.

Chapter 1. Introduction

1.1. Why ProntoScript?

ProntoScript allows one to add flexible 2-way communication and dynamic UI's to the Pronto system, bringing an ever higher level of home automation sophistication.

It is a system that:

- has easy to use plug-and-play modules for the custom installer
- is powerful and flexible for the 2-way module programmer
- is easy to learn

It is based on JavaScript, a popular and proven scripting language. Integrated into ProntoEdit Professional, it unlocks the full power of the WiFi-enabled Prontos and Extenders:

1. JavaScript is a modern, very high level programming language, allowing rapid development of rich end user applications
2. The web offers plenty of references and solutions to general programming challenges in JavaScript, more than any other language.
3. Encapsulated into a single Pronto Activity (Device), that can be merged into projects, the complexity of the code can be shielded completely from the custom installer. He just wants to plug in a 2-way module for controlling his selected equipment.

A few standardized hidden pages with instructions and parameters allow him to configure the module to operate seamlessly within his specific system.

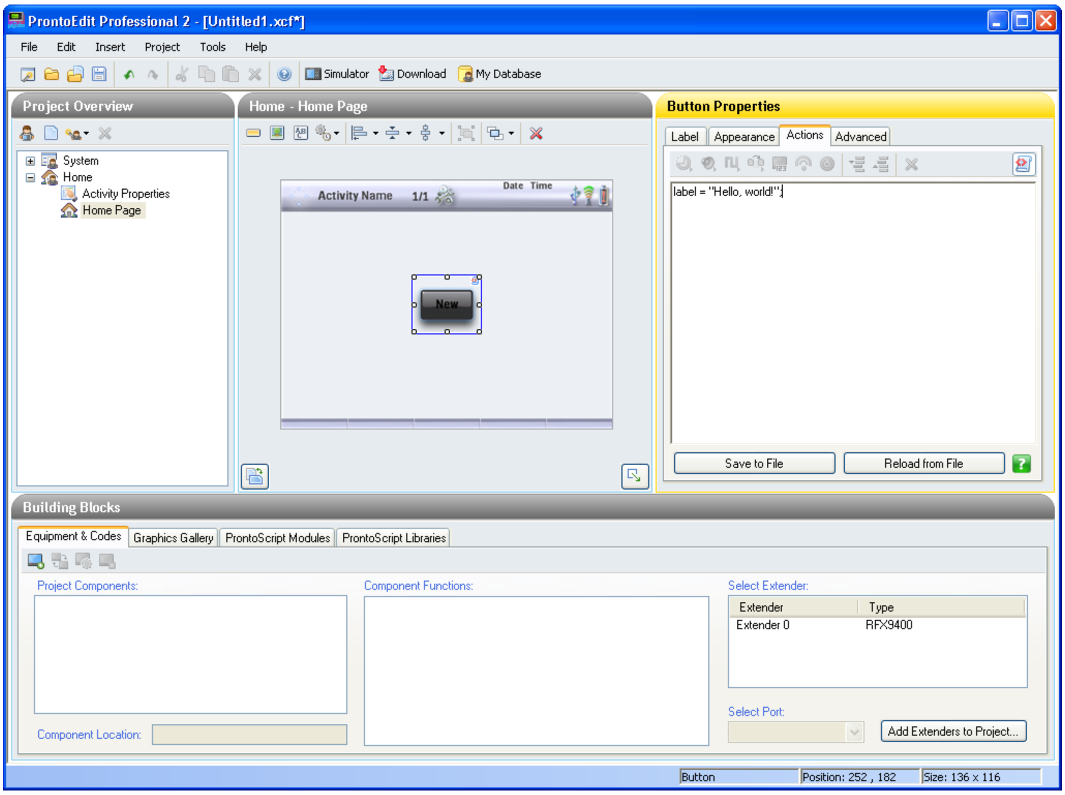
Let's begin with the classic "Hello, world!" program and see how to write this in ProntoScript.

1.2. A simple button script

Example 1.1. Simple button script source code

```
label = "Hello, world!";
```

By specifying the above ProntoScript for a button, its label will be changed to the famous greeting message at the moment a button is pressed.



```
label = "Hello, world!";
```

Extender	Type
Extender 0	RFX9400

To try out this example:

- Open ProntoEdit Professional 1.1 or above
- Create a new configuration (**Ctrl+N**)
- Open the home page and add a button to it (**Alt+B**)
- In the Button Properties, in the Actions tab:

  a. Press the 'PS' toolbar icon

  > **Note**
  >
  > Starting with ProntoEdit Professional 2.3, it is possible that a message "Pronto Script code can not be seen in standard viewing mode" appears.
  >
  > If this is the case, select the Options... menu entry of the Tools menu in the main menu bar, to show the Options dialog. In that dialog, select the General Settings tab, and select Advanced view in the View Mode Selection box.

  b. Add the ProntoScript code as shown

- Download to the Pronto (**Ctrl+D**)
- On the Pronto, press the button you created once

1.3. ProntoScript features

The main features of ProntoScript are:

- ProntoScript is based on JavaScript 1.6

- The ProntoScript API exposes a set of objects that represent the Pronto System, the Graphical User Interface and the Extenders.
- ProntoScript is embedded in the UI of the ProntoEdit Professional, facilitating writing and testing custom code for the Pronto.
- ProntoScript based 2-way modules can be integrated into any new or existing Pronto configuration project by means of the merge feature.

ProntoScript is based on the popular JavaScript scripting language, as used in Internet web browsers. In fact, the core ProntoScript language is largely compatible with ECMAScript-3, as present in popular web browsers such as Microsoft's Internet Explorer, or Mozilla Corporation's Firefox.

Think of any programming challenge you faced in the past with languages like C, Pascal, C++: with JavaScript (ProntoScript) you'll be able to handle it too, but most probably with less lines of code (and less hassle). This is illustrated with the examples in the following chapters.

JavaScript has a top notch arsenal of powerful tools for data processing, so much needed to write state-of-the-art 2-way communication drivers for a 2-way controller like Pronto.

Most RS-232 and TCP based protocols are ASCII based, some of them XML based. JavaScript provides two powerful tools for tackling those: regular expressions and ECMAScript for XML (E4X).

1.3.1. Regular expressions

Regular expressions allow you to take any kind of data stream input and filter it for the information that you need: either to update the display or know the exact 'state' of the equipment you are communicating with.

Example for a volume change response of an A/V receiver:

MV80<*CR*>

or in JavaScript:

```
var response = "MV80\r";
```

To filter out the integer value 80 without relying on the fact that it is exactly 2 characters starting at the position 3 one could use:

```
var volume = parseInt(response.match(/\d{2,}/)[0]);
```

With this one line of code, volume will hold the correct volume value even if the response would (hypothetically) be: "`%^&\r MV#80\r`".

This would not be possible with a simple substring operation.

Regular expressions, although a bit cryptic, are really great for Pronto communication jobs.

1.3.2. E4X

E4X is a recent addition to JavaScript to reference the increasing amount of internet data that is presented in XML format. If your Custom Install equipment communicates with XML, then parsing that data becomes an order of magnitude easier with E4X than it would be with classic regular expressions.

The XML processing support available in ProntoScript is specified in the [ECMA357] standard.

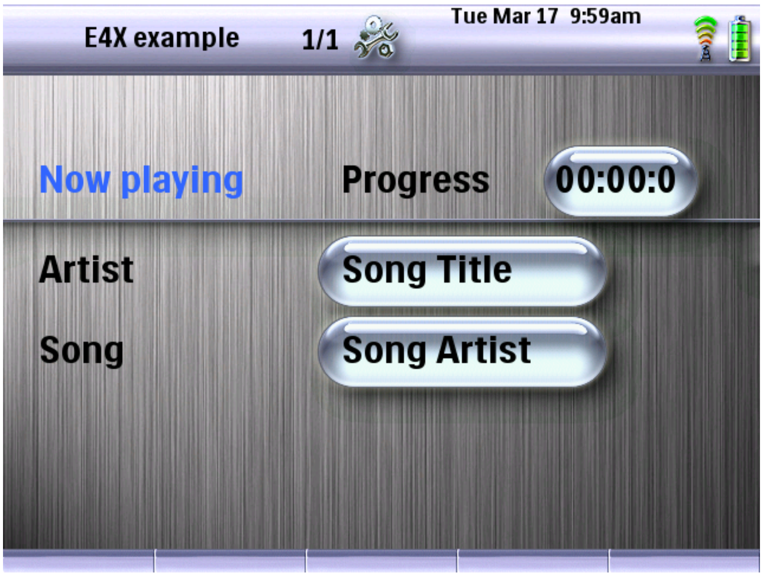
Example 1.2. Processing data in XML format

```
var incomingdata =
<body>
    <content id="200" title="Now Playing" bg="50" bgfit="s"
menuid="1000">
        <txt align="c" wrap="0">Title: <em>Song Title</em></txt>
        <br/>
        <txt align="c" wrap="0">Artist: <clr rgb="F0F0F0">Song
Artist</clr></txt>
        <br/>
        <txt align="c" wrap="0" rgb="0F0F0F">00:00:00</txt>
        <br/>
        <img align="l" id="16" alt="[Album Cover]" />
    </content>
</body>;
```

Then these 5 lines of ProntoScript code will parse it and show the correct information on the screen of the control panel:

```
var body = incomingdata;  // <body>...
GUI.widget("PLAYING_STATUS").label = body.content.@title;
GUI.widget("SONG_TITLE").label     = body.content.txt[0];
GUI.widget("ARTIST_NAME").label    = body.content.txt[1];
GUI.widget("PROGRESS").label       = body.content.txt[2];
```

The result could look like this:



Note

If the XML data is stored as a string (for example, because it was obtained from a TCP socket), it first needs to be converted to an XML object, before it can be accessed as XML:

```
var incomingdataText =
"<body>" +
"<content id=\"200\" title=\"Now Playing\" bg=\"50\" bgfit=\"s\"
menuid=\"1000\">" +
"<txt align=\"c\" wrap=\"0\">Title: <em>Song Title</em></txt>" +
"<br/>" +
"<txt align=\"c\" wrap=\"0\">Artist: <clr rgb=\"F0F0F0\">Song
Artist</clr></txt>" +
"<br/>" +
"<txt align=\"c\" wrap=\"0\" rgb=\"0F0F0F\">00:00:00</txt>" +
"<br/>" +
"<img align=\"l\" id=\"16\" alt=\"[Album Cover]\" />" +
"</content>" +
"</body>";
var body = new XML(incomingdataText);
GUI.widget("PLAYING_STATUS").label = body.content.@title;
```

The exact working of the statements used in the above script will be explained in the next chapters.

Chapter 2. Core JavaScript

This chapter describes the Core JavaScript features, which ProntoScript shares with other JavaScript-based environments, such as those found in web browsers.

2.1. Variables

The following examples tell you almost everything there is to know about variables in JavaScript:

```
var a = 10;              // declare a and assign integer value 10
b = "Hello, world!";     // declare b and assign a string
                         // (var is added implicitly)
b = 5;                   // JavaScript is untyped: b is converted
                         // automatically to hold an integer.
```

If you like more details, please refer to the [Flanagan] book or the [Mozilla] website: http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Variables

2.1.1. Primitive types

JavaScript has 3 primitive types: numbers, strings (of text) and booleans, plus two trivial datatypes: `null` and `undefined`.

2.1.1.1. Numbers

JavaScript does not distinguish between integers and floating points: all numbers are 64-bit floats.

Here are some examples of numeric literals:

```
var a = -10000; // integer literal
a = 0xff;       // hexadecimal literal (decimal 255 :-)
a = 1.797e-308; // floating point literal (e can also be E)
```

2.1.1.2. Strings

A string is a sequence of Unicode characters.

JavaScript is very flexible and powerful in working with strings, by means of automatic concatenation and number conversion. Some examples:

```
msg = "Hello, "+ "World!"; //msg -> "Hello, World!"
var a = 18;
hex_string = "0x" + a.toString(16); //hex_string -> "0x12"
var n = 12345.6789;
n.toFixed(0); //"12346"
n.toFixed(2); //"12345.68"
n.toExponential(2); //"1.23e+4"
n.toExponential(4); //"1.2346e+4"
n.toPrecision(3); //1.23e+4"
n.toPrecision(6); //"12345.7"
```

Some more examples on converting strings to numbers:

```
var division = "8" / "2"; // division is the number 4
parseInt("3 apples");     // returns to 3
parseFloat("3.14 kg");    // returns to 3.14
parseInt("0xFE");         // returns 254
```

2.1.1.3. Boolean

As in other programming languages, the boolean type is typically used for representing the result of comparisons, e.g. in an if-then-else statement.

Again JavaScript is not strict in types here and converts easily between boolean, number and string when appropriate: The boolean literals `true` and `false` are converted to `1` and `0` if used in a numeric context and to the strings `"true"` and `"false"` in a string context.

This means that people used to classic C programming can opt for `1` and `0` to represent On/Off states of custom install equipment. To advocate a consistent style however, we recommend using the boolean type explicitly:

```
var hallWayLights = false; //Hall Way Light Load, default is OFF
...

hallWayLights = getLightStatus();
if (hallWayLights) {
    // Hall Way Lights are ON
    ...
} else {
    // Hall Way Lights are OFF
    ...
}
```

2.1.2. Arrays

```
var a = new Array();
a[0] = 5;
a[1] = "Hi";
a[2] = { num:5, str:"Hi" }; //object with two properties num and str

var matrix = [[1,2,3],[4,5,6],[7,8,9]];
```

As in other languages, JavaScript offers arrays to store a collection of values into one object, which can be retrieved by a numeric index. The index always starts at `0`. Again, being untyped, the type of these values does not need to be the same for the different values as you can see in the examples.

As a result, Array size allocation is dynamic.

```
var a = new Array(5);
```

This creates an array with 5 undefined elements, but it cannot know yet, how much memory to reserve. Also, extra elements can be added by just assigning a value to it:

```
a[10] = "abc";
```

This extends the array to hold 11 elements.

2.2. Operators

JavaScript’s operators are inspired by the syntax of the C - C++ - Java language family.

For people with experience with these there are few surprises. This will be illustrated with a some examples.

2.2.1. Arithmetic operators

```
a = 5 + 6;          // a==11
a = 5 * 6;          // a==30
a = 5 / 2;          // a==2.5 !! all numbers are floats !!
a = parseInt(5/2);  // a==2
a = 5 % 2;          // a==1 (modulo, or remainder after division)
i = 1;
a = i++;            // a==1 i==2
j = 1;
a = ++j;            // a==2 j==2
```

2.2.2. Comparative operators

JavaScript supports =, == and === operators. These can be confusing to novice programmers:

2.2.2.1. Assignment Operator =

```
a = 5;
```

This is not a comparison operator, it is an assignment of the right-hand value to the left-hand variable.

Note

Please note this common C-language pitfall, which is also possible in JavaScript.

```
a = getLightStatus() // returns boolean true or false
if (a = true) {
   myLabel.label = "Lights are On";
} else {
   myLabel.label = "Lights are Off";
}
```

The programmer wanted to write:

```
if (a == true)
```

but forgot one '='. Instead of giving a warning or error, JavaScript will just assign `true` to `a`, and evaluate the assignment as always `true`.

So the test will always succeed, even if `getLightStatus` returned `false`.

2.2.2.2. Equality Operator ==

This is the operator that is used to compare for equality. Again, since JavaScript is untyped, it will use a "relaxed" form of "sameness" that allows type conversion

```
a = getLightStatus() // returns boolean true or false
if (a == "1") {
    ...
}
```

This will give the result the programmer intended, as `"1"` and `a` will be converted to the number `1` and then successfully compared.

In most cases, this relaxed comparison is sufficient. However, the automatic type conversion can lead to subtle bugs; to avoid these, use the Identity operator (===).

2.2.2.3. Identity Operator ===

`true === "1"` will evaluate to `false` as both are not identical because they are not of the same type.

The most practical use is if you really want to distinguish between `undefined` (declared but never assigned a value) and `null` (not a valid object)

```
var a = new Object;

myLabel.label  = (a.b===undefined);  // evaluates to true

a.b = null; //or a.b = someFunction() that returns null

myLabel.label  = (a.b===undefined); //evaluates to false
myLabel.label  = (a.b===null);      //evaluates to true
```

2.2.3. Bitwise operators

Bitwise operators require integers, so JavaScript will implicitly convert numeric values to 32-bit integers before proceeding.

```
//Bitwise AND
0x1234 & 0x00FF // -> 0x0034: used typically for masking

//Bitwise OR
  0x02 | 0x8 | 0x10
//0000 0010 | 0000 0100 | 0001 0000 -> 0001 1010
//use to set bit field registers

//Bitwise NOT
~0x0f  // -> 0xfffffff0 or -16, typically used for flip-flops
```

2.3. Statement blocks

Statement blocks or compound statements are formed by adding curly braces around a set of statements. It allows you to add multiple statements in constructions where only one statement is allowed:

```
{
    a = 5;
    b = 6;
    c = a + b;
}
```

2.4. Control flow

For controlling the flow of program execution, JavaScript has the following set of constructs:

- if/else
- switch
- while and do/while loop
- for and for/in loop
- break and continue statements

2.4.1. if/else

```
if (expression)
  statement 1
else
  statement 2
```

The last two lines above are optional.

Example 2.1. `if/else`

```
if (counter > 5) {
    // counter limit reached
    ...
} else {
    counter = counter + 1;
}
```

2.4.2. switch blocks

```
switch (expression) {
  case value:
    statements
  break;
    case value:
      statements
      break;
    default:
      statements
      break;
}
```

Example 2.2. `switch` block

```
var dayName;
switch (dayNumber) {
  case 0:
    dayName = "Sunday";
    break;
  case 1:
    dayName = "Monday";
    break;
  case 2:
    dayName = "Tuesday";
    break;
  case 3:
    dayName = "Wednesday";
    break;
  case 4:
    dayName = "Thursday";
    break;
  case 5:
    dayName = "Friday";
    break;
  case 6:
    dayName = "Saturday";
    break;
  default:
    dayName = "Unknown";
    break;
}
```

Note

The JavaScript version of the `switch` statement is more flexible than in classic languages: the expressions used between the () and after case, can be of any form and type. They are evaluated and compared at runtime. It also means that they execute less efficiently than compile time versions of C, C++ and Java.

2.4.3. while loops

```
while (expression)
  statement
```

Example 2.3. `while` loop

```
var i = 0;
while (i < 10) {
  i++;
  Diagnostics.log(i);
}
```

2.4.4. for loops

```
for (initialize ; test ; increment)
  statement
```

Example 2.4. `for` loop

```
var i;
for (i = 0; i < 10; i++) {
  Diagnostics.log(i);
}
```

```
for (variable in collection)
  statement
```

Example 2.5. `for` / in loop

```
var messages, i;
messages = [ "one", "two", "three" ];
for (i in messages) {
  Diagnostics.log(i);
}
```

2.4.5. break statement

The `break` statement causes the execution flow to exit the enclosing loop or `switch` statement.

2.5. Exceptions

Explicit exception handling is a proven technique to keep robust code simple and easy to maintain. You do this by separating the code that references error cases from the regular flow of the application.

A relevant example is to reference the possible exception you get when executing a Pronto button action list in an asynchronous timer callback. Only one action list can be executed at a time and it is possible the user just pressed a button when the timer expired.

Example 2.6. Exception

```
Activity.scheduleAfter(1000, timerTick);

function timerTick()
{
    try {
        CF.widget("MY_BUTTON","MY_PAGE").executeActions();
    } catch (e) {
        Diagnostics.log("System Busy executing actions");
    } finally {
        Activity.scheduleAfter(1000, timerTick);
    }
}
```

2.6. Functions

```
function funcname([arg1 [,arg2 [..., argn]]]) {
  statements
}
```

In JavaScript, functions serve several purposes:

- Define a chunk of functionality but don't execute it yet.
- Execute it at a later stage by calling the function.
- Encapsulate logic into organized, reusable blocks.
- Change the behavior of a particular function by passing parameters (arguments) to it
- Speed up execution as the function is compiled once, when it is defined: it does not need to be recompiled.
- Advanced: as a closure, to restrict the scope of a variable. (JavaScript does not have block scope, so a function is the only way to have "private" variables)
- Advanced: allow the programmer to write (pseudo) classes for OO programming.
- Advanced: register the function reference as an asynchronous callback, to be executed by the system at a later stage.

2.7. Objects

An object is a collection of named values, called properties. The ProntoScript API offers many useful objects to the programmer

```
var myButton, myButtonText;
myButton = GUI.widget("MY_BUTTON");
myButtonText = myButton.label; // use the label property
                               // of the button class
```

You can also define your own objects.

This is useful as objects allow you to better structure your code by encapsulation: grouping data and functionality that logically belong together into a single object.

Example 2.7. Custom `Object` creation

```
var myReceiver  =  new Object();
myReceiver.brand = "MyBrand";
myReceiver.model = "MyModel";
myReceiver.masterVolume = 60;
myReceiver.source = "DVD";
myReceiver.volumeUp = function() { this.masterVolume++; };
myReceiver.volumeUp();
myPanel.label = myReceiver.masterVolume; // shows 61
```

2.8. Built-in functions

Built-in Core JavaScript functions

`decodeURI`

Decodes a URI (such as the URLs used for HTTP resources), replacing the escape sequences used in them with the actual characters those escape sequences represent.

`decodeURIComponent`

Decodes a URI component; analoguous to `decodeURI`, but intended for URI components, not entire URIs

`encodeURI`

Encodes a string as an URI, replacing characters not allowed in a URI with escape sequences.

`encodeURIComponent`

Encodes a string as an URI component, replacing characters not allowed in a URI component with escape sequences. Similar to `encodeURI`, but also escapes the `:`, / and # characters which delimit components of a URI.

`escape`

A deprecated method to encode a URI. Use `encodeURI` or `encodeURIComponent` instead.

`eval`

Compiles and executes a string as a script, returning the result of that script.

`isFinite`

Tests if a value can be converted to a number, and is not negative or positive infinity.

`isNaN`

Tests if a value can be converted to `Number.NaN`, which is a special value representing and illegal number.

`parseFloat`

Interpret a value as a floating-point number.

`parseInt`

Interpret a value as an integer number.

`unescape`

Decodes values generated with the `escape` method.

`uneval`

Converts a function to a string.

2.9. Built-in classes

Built-in Core JavaScript classes

`Array`	Class implementation of an array.
`Boolean`	Representation of a boolean value.
`Date`	Representation of a date/time instance; in ProntoScript, this time instance is not the same as the user-visible time (which can be adjusted on the fly), but instead provides a monotonic clock, which can be used for timers and timeouts.
`Error`	Generic error exception class
`EvalError`	An error thrown for dynamic script evaluation failures.
`Function`	Class implementing functions, which in JavaScript are a special kind of objects.
`Math`	Provides various mathematical functions and constants.
`Number`	Representation of a numeric value.
`Object`	Base object class, from which all other classes and objects are derived.
`RangeError`	An error typically thrown when parameters are outside of an allowed range.
`ReferenceError`	An error typically thrown when a variable is used which is not defined.
`RegExp`	Representation of a regular expression.
`String`	Representation of a text string.
`SyntaxError`	An error thrown by the JavaScript engine when a syntax error is encountered.
`TypeError`	An error typically thrown when some value is not of the expected type.
`URIError`	An error thrown when a failure is encountered while processing a URI.

Built-in E4X JavaScript classes

`Namespace`	Representation of an XML name space, which can be used to obtain elements from an `XML` object which are not in the document's default namespace.
`QName`	Representation of a qualified XML name, as obtained using the `name` method of an `XML` object instance.
`XML`	Representation of an XML document or fragment.

2.9.1. Regular Expressions

See chapter 11 in [Flanagan]. Also a lot of information and examples can be found on the Internet.

2.9.2. Math object

The `Math` object gives access to a number of useful mathematical constants and functions.

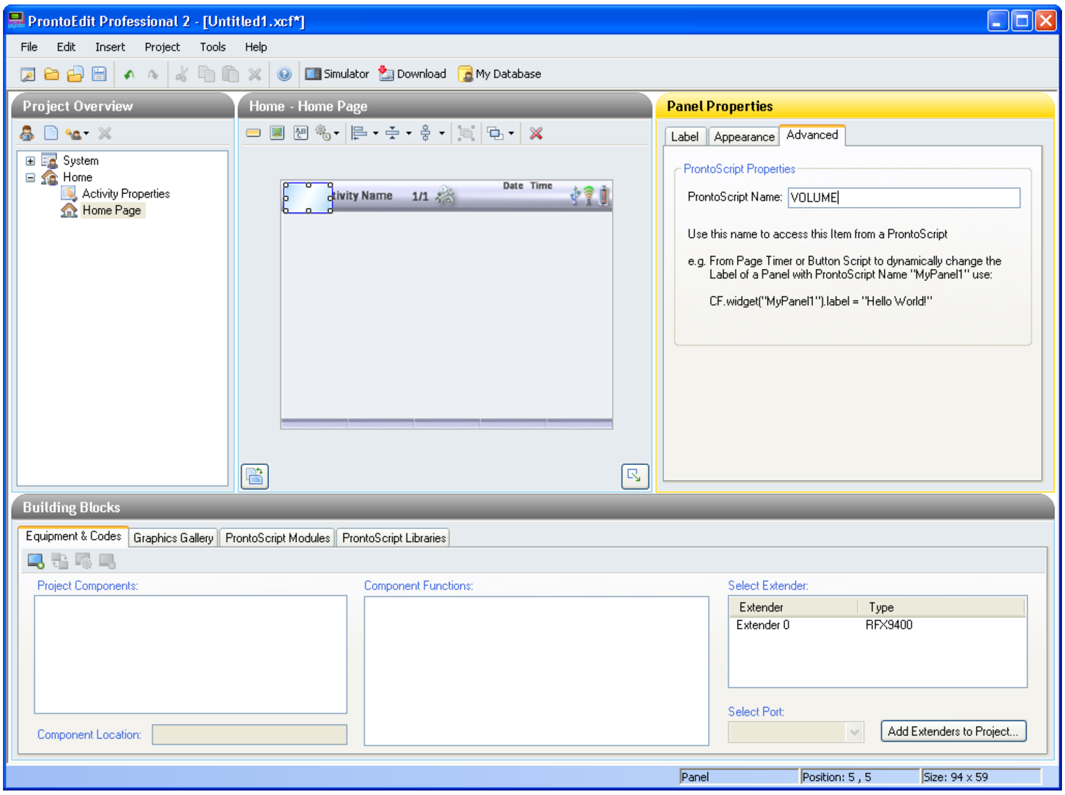
Example 2.8. `Math`

```
Math.floor(2.5); // -> 2
Math.ceil(2.5); //-> 3

Math.abs(-3); //-> 3

Math.random(); //->(pseudo)random number between 0.0 and 1.0

Math.PI; // -> 3.141592653589793
```

Chapter 3. Widgets

In the editor a page is composed of a number of graphical objects, called widgets. These widgets can be manipulated from a script to create a more dynamic user interface. The most obvious widgets are buttons and panels, but also hard buttons are considered as widgets because they share a number of properties with (soft) buttons.

All widgets have a tag, which is a unique identification string that is needed in order to get access to it from a script. Imagine you created a new configuration file with one panel on the home page: by default, it will display a white rectangle. Now, change its tag by putting the text "VOLUME" (without quotes) in the ProntoScript Name field in the Advanced tab of the property dialog for the panel:



ProntoEdit Professional 2 - [Untitled1.xcf*]
File Edit Insert Project Tools Help
Simulator Download My Database
Project Overview
System
Home
Activity Properties
Home Page
Home - Home Page
Activity Name 1/1
Date Time
Panel Properties
Label Appearance Advanced
ProntoScript Properties
Building Blocks
Equipment & Codes Graphics Gallery ProntoScript Modules ProntoScript Libraries
Project Components:
Component Functions:
Component Location:
Select Extender:
Select Port:
Add Extenders to Project...
Panel Position: 5 , 5 Size: 94 x 59

ProntoScript Name: VOLUME

Use this name to access this Item from a ProntoScript

e.g. From Page Timer or Button Script to dynamically change the Label of a Panel with ProntoScript Name "MyPanel1" use:

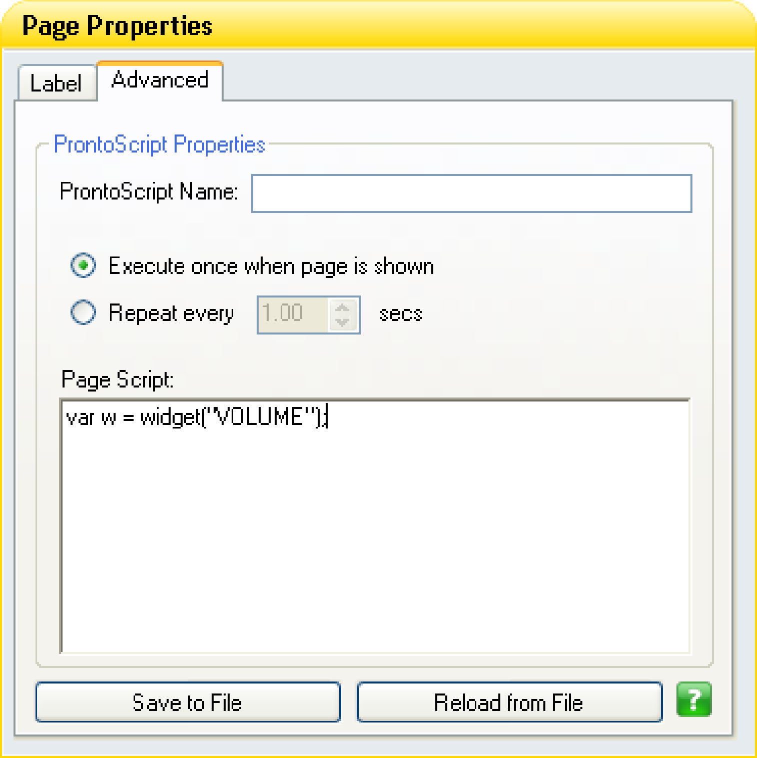
CF.widget("MyPanel1").label = "Hello World!"

Extender	Type
Extender 0	RFX9400

Warning

Often the tag or ProntoScript Name of a widget is confused with the label. Remember that the tag is the invisible name of the widget and the label is the text that is displayed in the widget.

In the page properties dialog of the Home page, go to the Advanced tab and in the Page Script input field, put the following line:



```
var w = widget("VOLUME");
```

This one line of code looks in the page for a widget with the tag "VOLUME". It finds the panel and stores a reference to it in variable w.

Note

The tag is case sensitive, so "VOLUME", "Volume" and "volume" are considered different tags! Therefore, try to be consistent when using uppercase and lowercase.

Tip

A recommended convention is to use UPPERCASE for tags and lowerCamelCase for variables.

Once you have a reference to a widget, you can manipulate its properties. The next paragraphs will show you some exciting examples for the different widget types.

3.1. Panels

The simplest widget type is a panel. Panels are a placeholder for text and/or an image. Until now you used it to display text somewhere on a page, or to put some nice graphics on the background. Now, with ProntoScript, the panels become dynamic. They can now show the state of the system, just like the special widgets, called System Items that you are used to seeing on the system page. For example the battery and WiFi widgets show a different image depending on an internal variable. The Activity Name widget shows the name of the current activity but sometimes also shows strings like "Connecting…" or "Command failed".

3.1.1. Change the label

The label can be used for example to show the amplifier volume, the current tuner frequency or the currently playing song title.

With the above page script, a reference `w` is retrieved, to a panel with the tag "VOLUME". In order to to show a real dynamic value on it, some code can be added to the page script to create a variable to contain this value:

```
var w, volume;
w = widget("VOLUME");
volume = 0;
w.label = volume;
```

When this is downloaded to the control panel, and the page is displayed, the panel will show "0" immediately. This is because the page script is executed already before the page is really displayed. So the labels of all widgets on a page can be properly initialized in the page script of that page.

Note

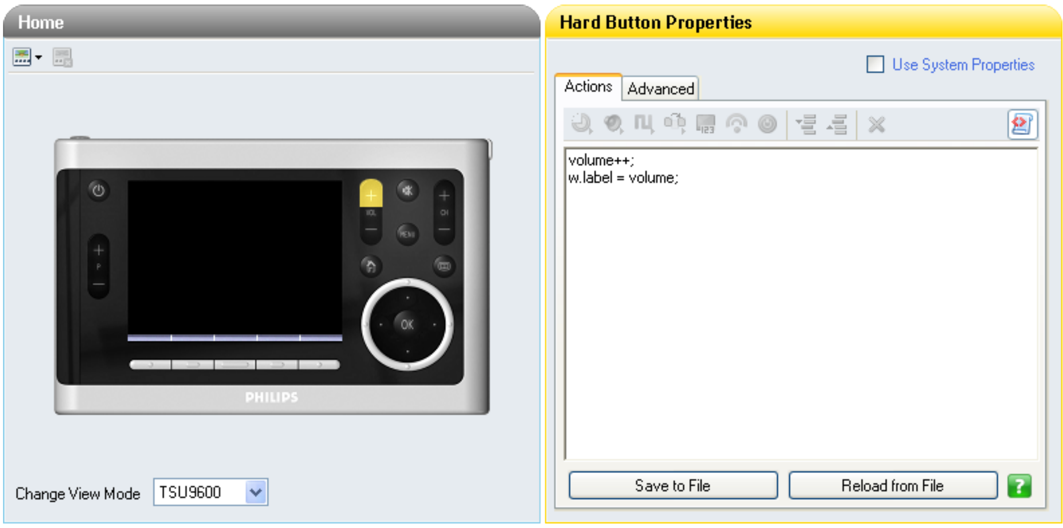
The volume, which is an integer number, is automatically converted to a string when assigning it to the label property of the widget variable `w`.

Now, let's change the volume. In the Home Properties, select the **VOL+** button.

Note

In ProntoEdit Professional 1.x, you first have to select the Hard Buttons tab before being able to select a hard button.

Unselect the "Use System Page Actions" checkbox. Then click on the ProntoScript icon to show the script input field. Add the following code:



□ Use System Properties

```
volume++;
w.label = volume;
```

This little script increments the volume variable and updates the label of the volume panel with the new value.

Note

`volume` and `w` do not need to be declared again in the button.

Variables that are declared in the page script can be accessed from all the button scripts on that page.

In the same way, the following code can be added to the **VOL-** button:

```
volume--;
w.label = volume;
```

Now download this to the control panel and play with the **VOL+** and **VOL-** buttons. You will see that the value displayed in the panel will count upwards and downwards accordingly.

3.1.2. Change the position

It is just as easy to change the position of the panel. Just change the value of the properties `top` and `left`. As an example, put the following code to the cursor arrow keys:

Cursor up:

```
w.top -= 10;
```

Cursor down:

```
w.top += 10;
```

Cursor left:

```
w.left -= 10;
```

Cursor right:

```
w.left += 10;
```

Download to the control panel and play with the cursor keys and see the volume walk around the screen. Confirm that you can move the panel completely off the screen.

3.1.3. Hide and show

You can hide and show the panel as you wish. The panel has a property called `visible` . When writing `true` or `false` to it, you are directly in control of its visibility. In the example, put the following script in the **OK** hard button:

```
w.visible = !w.visible;
```

The not (`!`) operator negates the value that comes after it. Can you predict what will happen when you press the ok button when you download this to the control panel?

3.1.4. Change the image

When you want to have a panel with dynamic graphics, you have two options.

The first option is to create a separate, hidden page in the same activity and attach each image you want to display to a separate panel in this activity. Give the hidden page a label and a tag, for example "`RESOURCES`". Give the panels tags like "`VOLUME0`", "`VOLUME1`", etc.

Now these images can be accessed:

```
function showVolume()
{
    var v;
    w.label = volume;
    v = widget("VOLUME"+volume, "RESOURCES");
    if(v) {
        w.setImage( v.getImage() );
    }
}
```

This code copies the image of one of the resource panels to the volume panel. Especially note the validity check on `v`: if the widget is not found, `v` will not be a valid widget reference and `v.getImage()` would throw an exception causing the script to be aborted. The `if(v)` makes sure the image is only copied when `v` is not `null`.

The `stretchImage` property can be used to automatically resize the images in a widget. In the above example, this can be done by adding the following line:

```
widget("w").stretchImage = true;
```

Remember that this is a property for a widget and not an image! So when you call this property once, all images in this widget will be rescaled to fit the widget size. If you do not set the `stretchImage` property, or set it to `false`, the image will not be scaled. This means that if your image is larger than your widget, a part of the image will not be visible. If the image is smaller than the widget, a part of the widget will be empty.

When you use `stretchImage` for a button, both the pressed state and the released state images are stretched if `stretchImage` is set to `true`. And the last thing you should know about `stretchImage` is that when you copy a stretched image, you will get the original image from the widget. So no data is lost!

The second option to make a panel with dynamic graphics is by dynamically creating the images. This allows you to get the images from a web server or even construct the images yourself, pixel by pixel!

In the example configuration above, in the button scripts for **VOL+** and **VOL-**, replace the line:

```
w.label = volume;
```

with the line:

```
showVolume();
```

3.1.5. Change the background color

The background color of a panel which does not have a background image can be controlled with the `bgcolor` property.

The value used to set the color is a hexadecimal representation of the blue, green and red color components (similar to the way colors are set in HTML web pages).

```
w.bgcolor = 0x0000ff; // Red
w.bgcolor = 0x00ff00; // Green
w.bgcolor = 0xff0000; // Light Blue
w.bgcolor = 0xffff00; // Turquoise
w.bgcolor = 0x00ffff; // Yellow
w.bgcolor = 0xef358e; // Purple
```

3.1.6. Change background transparency

The transparency of a panel which does not have a background image can be controlled with the `transparent` property. Setting this property to `true` causes the panel background to be transparent. Setting this property to `false` causes the panel background to be opaque.

In the example, put the following script in the **OK** hard button:

```
w.transparent = false;
```

Pressing OK will now cause the panel *w* to be rendered with an opaque background.

3.2. Buttons

Buttons are put on a page to create a clickable area. So, you created a button, attached two images to indicate its released and pressed state and gave it a label to be displayed on it. And, of course, you attached actions to it. This is as far as you could go with the traditional Pronto buttons. With ProntoScript, there is a lot more which can be done with buttons, as shown below.

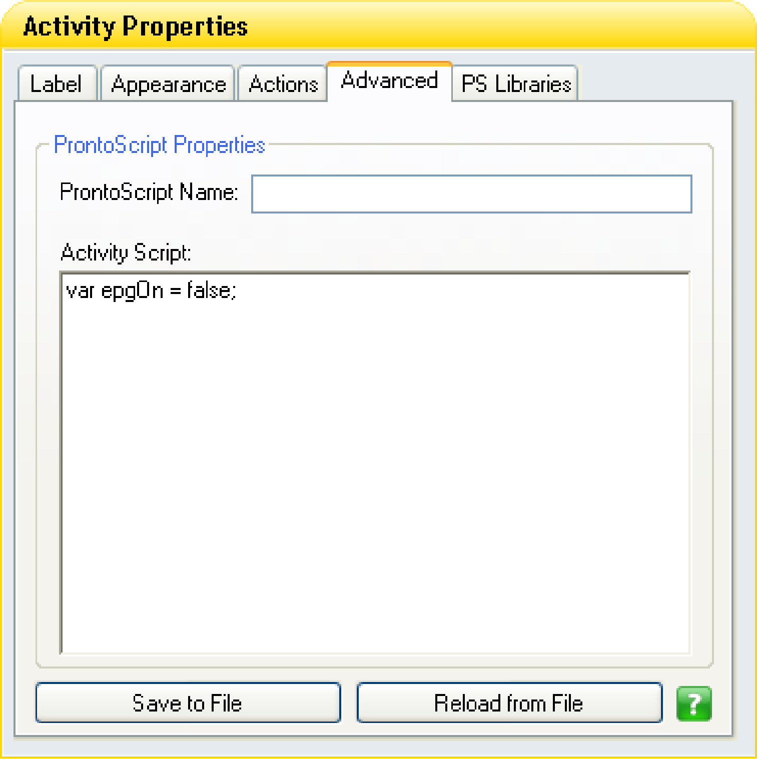
First, a tag should be given to the button, for example "MY_BUTTON" so it can be retrieved from the page script:

```
var w = widget("MY_BUTTON");
```

3.2.1. Toggle button

A toggle button is a button that can show two (or more) states. For example, in your "Watch TV" activity, you want to remember if you entered EPG mode or not.

First, create an activity variable to hold this state by declaring it in the activity script:



ProntoScript Name:

Activity Script:

```
var epgOn = false;
```

Then, when the page with the button is displayed, it should be properly initialized in the page script:

```
var w = widget("MY_BUTTON");
w.label = epgOn ? "On" : "Off";
```

This locates the button and then gives it the label "On" when `epgOn` equals `true`, and "Off" otherwise.

Now, in the button script, some code can be put to toggle the state and show it:

```
epgOn = !epgOn;
label = epgOn ? "On" : "Off";
```

Note

The reference `w` is not needed here to get to the label because this script is executed within the scope of the button object. For more information on button scope, refer to Section 6.2, “Scope”.

3.2.2. Info popup

Suppose you want a popup window to be displayed for as long as you press a button. This can be done by defining an `onRelease` function. Create a panel with the desired image and text and give it a tag "`INFO`". In the page script, get a reference to the panel and make it invisible by default:

```
var info = widget("INFO");
info.visible = false;
```

Then, let's program the **GUIDE** hard button with a little script to show the info panel when it is pressed:

```
info.visible = true;
onRelease = function()
{
    info.visible = false;
};
```

3.2.3. While-pressed counter

If you want an action to be repeated for as long as a button is pressed, you can define an `onHold` function in the button script. Also set the `onHoldInterval` property to the number of milliseconds between two repeats:

```
var counter = 0;
onHold = function() {
    counter++;
    label = counter + " seconds";
};
onHoldInterval = 1000; // msec
```

Download this to the control panel. Then, press the button and keep it pressed. Do you see the label counting the seconds? What happens after 30 seconds? It will stop counting! This is because of a safety mechanism built into the Pronto software. If it detects a button being pressed (stuck) for more than 30 seconds it stops the associated action.

3.3. Hard buttons

The hard buttons are different from the buttons described above in the sense that they do not have any graphical properties like label, image, visible, etc. What you can do however is to define some `onHold` or `onRelease` functionality for them. In order to get access to the hard buttons, some predefined tags are available. See Appendix C, *Predefined tags* for the full list.

3.4. Firm keys

Firm keys are the five hard buttons on the bottom of the LCD display with the corresponding buttons right above them. They have an image, a label, position etc. just like other buttons, but they are special. The editor does not allow you to define a tag for them. Instead, you can get access to them using the predefined tags `PS_FIRM1` etc.

In the editor you can only define the firm key behavior on activity level, so normally the firm keys are the same for all pages in one activity. Scripting allows you however to make them look different on each page by changing their labels or even their images or position in the page script:

```
var firm1 = GUI.widget("PS_FIRM1");
firm1.label = "Blabla";
function onFirm1()
{
    ... // put here your firm key code
}
```

And then put the following script in the firm key on activity level:

```
onFirm1();
```

For an extensive list of all the Widget properties and methods, please refer to Appendix A, *ProntoScript Classes Description (ProntoScript API)*.

Chapter 4. Action Lists

One thing all widgets except panels have in common is that you can define a list of actions for them in the editor. This includes sending infrared codes, performing page jumps, playing of sounds, etc.: a lot of interesting stuff you also might want to do from ProntoScript. The `executeActions()` method of widget objects (such as buttons), can be used to execute these actions.

For example, a button can be created that sends the infrared codes only when the button is pressed for at least one second by putting the following code in its script:

Example 4.1. `onHold`

```
onHold = function()
{
    executeActions();
};
onHoldInterval = 1000; // msec
```

This example first defines an `onHold` function that invokes the action list. This function then is scheduled after one second.

Another example is the EPG toggle button that sends different infrared codes to enter and exit EPG mode. For this, generate two buttons with the different infrared codes, tag them "`EPG_ON`" and "`EPG_OFF`" and put them on a separate page tagged "`IRCODES`". Then adjust the toggle button script to do the trick:

Example 4.2. `executeActions()`

```
epgOn = !epgOn;
label = epgOn ? "On" : "Off";
page("IRCODES").widget(epgOn ? "EPG_ON" : "EPG_OFF").executeActions();
```

Note

Action lists can not be executed in parallel. This means that when a script calls the `executeActions()` method while an action list is already being executed currently, an exception will be thrown.

When the calling script requires the action list to be executed, it is advised to schedule a new `executeActions()` in the handling of this exception. See example in Section 2.5, “Exceptions”.

Tip

Calling `executeActions()` will not work from within a page script. This is due to the restriction that the control panel cannot play multiple action lists at the same time and a page script is always executed after a page jump action within an Actionlist

The way to do this is by scheduling the playing of the action list as follows:

```
...
scheduleAfter(100, function(){ executeActions(); });
...
```

Chapter 5. Timers

ProntoScript provides three mechanisms which can be used for delaying execution of scripts:

- Fully blocking waits with the `delay()` method,
- page timers, and the
- `scheduleAfter()` method.

5.1. Blocking wait

Sometimes you need some time between two script statements. The `System.delay()` function can be used for that. Just pass the desired number of milliseconds as a parameter. For example, you want a button that turns on the hallway light and automatically turns it off after 10 minutes. You can do this with the following button script:

Example 5.1. Blocking wait

```
page("IRCODES").widget("HALL_LIGHTS_ON").executeActions();
System.delay(10*60*1000); // msec
page("IRCODES").widget("HALL_LIGHTS_OFF").executeActions();
```

Download this to your panel, press the button and sit back and wait… This should block the control panel for a full 10 minutes.

Note

When executing this script, the screen of the control panel looks frozen. The control panel will not respond to any key presses during the 10 minute delay.

Hence, using `System.delay(..)` is typically not a good idea for user-noticable delays (more than 100 milliseconds).

5.2. Page timer

The editor allows you to mark a page script as repetitive. This feature can be used to count down until it is time to turn off the lights.

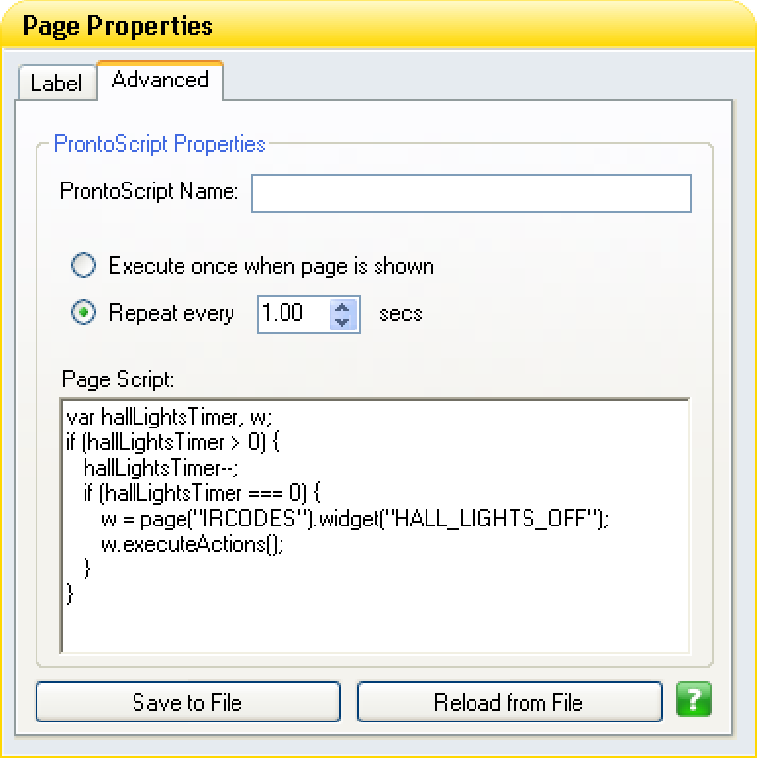
First declare a counter in the activity script:

```
var hallLightsTimer = 0;
```

Then, in the button script, turn on the lights and start the timer by setting the counter. Setting it to 10 seconds instead of 10 minutes allows for quicker testing:

```
var w = page("IRCODES").widget("HALL_LIGHTS_ON");
w.executeActions();
hallLightsTimer = 10; // seconds
```

Finally define a page script to be called every second to decrement the counter and turn off the hall lights if the counter reaches zero:



ProntoScript Name:

○ Execute once when page is shown

● Repeat every 1.00 secs

Page Script:

```
var hallLightsTimer, w;
if (hallLightsTimer > 0) {
  hallLightsTimer--;
  if (hallLightsTimer === 0) {
    w = page("IRCODES").widget("HALL_LIGHTS_OFF");
    w.executeActions();
  }
}
```

Now try this on your control panel. Do you notice that the control panel is fully operational while the timer is running? In fact, you don't even notice it. Except for the hall lights being on, you have no indication that the page script in fact is activated every second. Maybe it is a good idea to show a little icon somewhere on the screen to indicate that a timer is running. Or maybe after each decrement of the counter update the button label with the remaining time:

```
hallLightsButton.label = hallLightsTimer + " sec";
```

This implementation using the page timer has a number of drawbacks: You normally want to put a lot of code in the page script but you probably don't want all of this code to be repeated continuously. There is only one page timer. If you need multiple timers you will need to use the `scheduleAfter()` function discussed in the next section.

5.3. scheduleAfter()

A more sophisticated way to reference the hall lights timing is to use the third timer mechanism: `scheduleAfter()` . This method of the Activity class allows you, as the API reference states in Appendix A: "to program a function to be executed once after a certain time". This requires a function which turns off the lights, and a call to `scheduleAfter()` to trigger it, as shown in the following button script:

Example 5.2. `scheduleAfter()`

```
page("IRCODES").widget("HALL_LIGHTS_ON").executeActions();
function hallLightsOff()
{
    page("IRCODES").widget("HALL_LIGHTS_OFF").executeActions();
}
scheduleAfter(10*60*1000, hallLightsOff);
```

5.4. Behavior during sleep mode

When the control panel is asleep, all timers are stopped. This includes the page timer and the scheduleAfter timer. When the control panel is woken up again, the timers are resumed.

There are two exceptions in which the control panel does not go to sleep:

- The control panel is put into the docking. While the control panel is powered there is no need to save battery consumption.

- The control panel is connected to a PC with a USB cable. In this case the control panel cannot go to sleep because it needs to respond to USB messages.

In these cases the screen of the control panel will be turned off, but the timers keep running as expected.

You can also configure the screen to be always on. To do this, enter settings by pressing and holding the settings icon for three seconds. Then on the second tab, increase the value below the text "Turn screen off after:" until it displays "On".

Chapter 6. Levels, scope and lifetime

6.1. Levels

A configuration file is a hierarchy of a number of activities or devices, each consisting of a number of pages, each having a number of buttons and panels. The editor shows this hierarchy in its tree view. You can attach scripts to all levels within this hierarchy: activity, page and button.

6.2. Scope

The preceeding chapters already covered several code snippets and mentioned scope once or twice. This section covers this subject in more detail.

6.2.1. Local scope

When you declare a function or variable in a button script, it will be known only in that script. That is called local scope.

6.2.2. Page scope

But when you declare something in a page script, it will be known in all the button scripts on that page. So you can declare a variable like `epgOn` in a page script and use it in a button script on that page. The other pages however cannot access this variable in any way: that is page scope.

6.2.3. Activity scope

Everything declared in an activity script is known in all the page scripts and button scripts of that activity. So if you declare a function like the `onFirm1()` at activity level you can call it from the firm key scripts in that activity but also from any of the page scripts of that activity as well as from any button script on any of the pages of that activity. But the function cannot be accessed from other activities and when you switch to another activity, all declarations and definitions are destroyed.

The advantage of this mechanism is that if you have two activities, they can use the same names in their scripts without interfering with each other. But sometimes you want to explicitly share information with other activities, or store some persistent data so that you can restore the state of the activity after switching to another activity and back.

For these cases, system globals can be used.

6.2.4. System globals

System globals allow information to be used by multiple activities. You can store a string globally using the `System.setGlobal()` method and retrieve it with the `System.getGlobal()` method.

6.3. Lifetime

The lifetime of a script object is the time that the function, variable or class remains defined after its declaration. This is defined by the time that the scope, in which the object is declared, remains active.

In ProntoScript, all scopes remain active as long as the activity remains active. This means that variables set in one page will still have their values retained when coming back to that page.

Chapter 7. Activities and Pages

A few examples of page and activity scripts have been covered in previous chapters. This chapter discusses these two script types in more detail.

7.1. Activity script

The activity script is executed when you 'enter the activity'. This means when a page of the activity is about to be displayed, and the previous page was not part of this activity.

The activity script is executed just after the activity is initialized, but before the page script is executed. The page objects are not created yet. (so never use `GUI.widget()` at Activity level!)

7.1.1. Usage

The activity script can be used to initialize an activity, to define objects, functions and variables that need to be used on all its page scripts. It also typically defines any parameters of the activity.

If functionality needs to be executed only the first time the activity is entered, a global variable can be declared to check whether the activity script is already executed or not. For example, you have an activity "Listen to iPod" and you want to initialize it the first time you connect to it:

Example 7.1. `System.setGlobal()` / `System.getGlobal()`

```
if (System.getGlobal("ListenIPod.Initialised") == null) {
    ... // perform first time initialisation
    System.setGlobal("ListenIPod.Initialised", "true");
}
```

Be aware that since the page objects are not created yet, it is not possible to show any feedback to the user here. This should be done in the page script.

7.1.2. Home activity

A special activity is the Home activity, since it is the first activity that is selected after the control panel is powered or after a configuration file download. The Home activity script should contain the definitions needed in all the home page scripts. Besides that, it can also be used to initialize the global variables stored in the `System` class.

Note

The editor does not allow renaming the Home activity? With ProntoScript you can (although it is not recommended). Just add this line to the Home activity script:

```
label = "Lobby";
```

7.1.3. Rotary wheel

Rotations from the rotary wheel can be handled in ProntoScript using the `onRotary` callback function. It allows you to capture the movement of the rotary wheel and take actions accordingly.

```
onRotary = function(clicks)
{
      //put your code here
};
```

To know in which direction the user turned the wheel, `clicks` is positive for clockwise and negative for anticlockwise rotations. To let the programmer know when the user stops turning the rotary wheel, the last value of `clicks` is always a single `0`.

7.1.4. Advanced rotary wheel example

With the ability to use the rotary in your ProntoScript you can easily browse through lists. A good example is controlling a music player. Of course, when there are a lot of items in the list, you don't want to scroll for too long before reaching the last item. So in this paragraph we'll show you how you can implement an acceleration algorithm by using a weighted exponential function, getting you even faster to the correct item in the list.

The acceleration algorithm makes use of some mathematical functions:

- `Math.abs` calculates the absolute value of its argument. (e.g. the result of `Math.abs(-3)` is 3)

- `Math.exp` calculates the exponential value of its argument. (e.g. `Math.exp(3)` equals e^3 and has as result 20,08)

To change the result of the `Math.exp` back to an integer value, the function `parseInt` can be used. (for example, the result of `parseInt(3.25)` is 3)

In the code below, the result of the function will be stored in the variable `accClicks`, standing for accelerated clicks. As you can see in the code the absolute value of clicks is calculated first, and stored in the variable `absClicks`. This allows us having the same code for both positive and negative values of clicks. But when doing this, the sign has to be put back at the end of the function!

In the first `if` statement, a test is performed to see whether the number `absClicks` the rotary moved is bigger than 2. This is done because if it is less, the user probably doesn't want to scroll very fast through the list. If the result is bigger than 2, exponential acceleration is used and the following value is calculated:

$$\frac{e^{(absClicks-1)}}{(absClicks-1)}$$

resulting in a smooth acceleration.

If the number of clicks was less than or equal to 2, the result is always changed to 1 (with respect to the sign of clicks). This way, the user is scrolling slowly through the list, just as he expects.

Example 7.2. `onRotary`

```
onRotary = function(clicks)
{
    var accClicks,
        absClicks,
        temp;
    absClicks = Math.abs(clicks);
    if (absClicks > 2) {
        temp = absClicks-1;
        accClicks = parseInt( (Math.exp(temp))/temp );

        if (clicks < 0) {
            accClicks = -accClicks;
        }
    } else {
        if (clicks > 0) {
            accClicks = 1;
        } else if (clicks < 0 ) {
            accClicks = -1;
        } else {
            accClicks = 0;
        }
    }

    /* use the accelerated clicks (accClicks) */

};
```

Of course, this is only one example of an acceleration algorithm. Many others exist, and it is the job of the programmer to find the right algorithm for his application.

7.2. Page script

The page script is executed just before a new page is going to be displayed. In fact, all the buttons and panels on the page are created as specified in the configuration file by the editor. The only thing that has not been done is to show them on the screen.

7.2.1. Usage

So now is the time to make some last-minute alterations! This means you can change the labels and images of widgets to show the actual status. And you can hide any popup panels and other widgets that should not be visible initially.

If you declared some general purpose functions in the activity script, you can use them here.

If you need some variables that need to be shared between different widgets on the page, you should declare and initialize them here.

7.2.2. Page label

In the editor you can define a label for every page. Without ProntoScript, these can not be used on the device.

The following actvity script animates the activity label and the page label:

Example 7.3. Label animation

```
var orgLabel = label; // Save original activity label

function animateLabel()
{
    if(label == "") {
        label = orgLabel; // Restore original activity name
    } else {
        label = label.substring(1); // Remove the first character
        scheduleAfter(330, animateLabel); // Animate 3x per second
    }
}

function startAnimateLabel(pageLabel)
{
    label = orgLabel + " – " + pageLabel; // Combine the activity and
                                          // page label
    scheduleAfter(2000, animateLabel); // Start animating after 2 seconds
}
```

Tip

It is a good practice to use comments to make complex scripts more readable as shown in the example above.

Now start the animation in the page script:

```
startAnimateLabel(label);
```

7.2.3. Home page

The home page is the first page of the Home activity. Since the home page is the first to be displayed after power up of the control panel, you can put a custom splash screen here. Create a panel with a nice background and a welcome message and tag it "SPLASH". Then, put the next code in the home page script:

```
if (System.getGlobal("Home.Started") == null) {
    scheduleAfter(3000, function() { widget("SPLASH").visible = false; } );
    System.setGlobal("Home.Started", "true");
} else {
    widget("SPLASH").visible = false;
}
```

7.2.4. Jump to another activity

When an action list containing a page jump to a page of another activity is executed, the lifetime of the current activity stops and the script is aborted. The execution of the action list however is not affected.

7.2.5. Multiple page jumps within an activity

When an action list containing multiple page jumps is executed, each page script is executed when the jump is done, and the next action in the action list is only executed after the page script has finished. This has the consequence that this page script can not execute an action list, since one is already being executed. An exception will be thrown. When the calling script requires the action list to be executed, it is advised to schedule a new `executeActions()` in the handling of this exception. See example in Section 2.5, “Exceptions”.

Chapter 8. Extenders

Now that you know how to create some scripts and to manipulate the widgets on the screen, it is time to interface with your equipment. This chapter covers the devices that you hooked up onto your serial extender(s); the next chapter will cover communicating to the rest of the world over the wireless network.

8.1. CF.extender[ ]

How to use an extender in ProntoScript? The CF class has a member called `extender[]` which is an array containing valid entries for all extenders that are configured in the editor.

Suppose you want to use an extender that you configured as extender 0. Then the following line gets a reference to the Extender object that corresponds to it:

```
var e = CF.extender[0];
```

If extender 0 is not defined, e will now have the value undefined, which is equal to `null`. If you want your script to protect against this, you can do it as follows:

```
if( e == null ) {
    Diagnostics.log("Extender 0 is not defined");
} else {
    ... // put the rest of your code here
}
```

The Extender object that you have now, gives you access to the ports of the extender: the serial ports, the power sense ports and the relay ports. It does this through its arrays: `serial[]`, `input[]` and `relay[]`.

Since a serial extender has four serial ports, four inputs and four relays, the arrays each contain four references to objects of type `Serial`, `Input` and `Relay`.

Note

Although the ports are numbered 1 to 4 on the extender and in the editor, all array elements start at index 0 in ProntoScript! This is according JavaScript convention.

8.2. Serial ports

Suppose you hooked up a serial A/V receiver onto the first serial port of the extender. First, a reference to that serial port must be obtained:

```
var s = e.serial[0];
```

If the extender is defined as a basic extender, it will have no serial ports and the entry will be `null`, so that can be checked against:

```
if( s == null ) {
    Diagnostics.log("Extender 0 is not a serial extender");
} else {
    ... // put the rest of your code here
}
```

8.2.1. Configuring the serial port

Once the `Serial` object for the serial port is retrieved, it can be configured with the serial communication settings that the receiver is expecting.

For example:

```
s.bitrate = 9600;
s.databits = 8;
s.parity = 0; // None
s.stopbits = 1;
```

These are in fact the default communication settings of the serial ports. But it is a good practice to explicitly configure them.

8.2.2. Sending and receiving

Now that the serial port is configured, a command can be sent to it to turn the A/V receiver on:

```
s.send("PWON\r");
```

This sends the string "PWON" followed by a carriage return over the serial line.

With the `receive` function, a command can be sent and a response received. This one line of code requests the current master volume:

```
var volume = s.match("MV?\r","\r",250);
```

This first sends the string "`MV?\r`" to the A/V receiver and then captures the incoming data until a carriage return is received. The last parameter makes sure the operation does not wait longer than 250 milliseconds for the response to be received.

Combining all above code snippets together yields a button script that requests the volume and puts it on its label:

Example 8.1. Synchronous serial communication

```
var e,s;
e = CF.extender[0];
if( e == null ) {
    Diagnostics.log("Extender 0 is not defined");
} else {
    s = e.serial[0];
    if( s == null ) {
        Diagnostics.log("Extender 0 is not a serial extender");
    } else {
        s.bitrate = 9600;
        s.databits = 8;
        s.parity = 0; // None
        s.stopbits = 1;
        label = s.match("MV?\r","\r",250);
    }
}
```

8.2.3. Asynchronous operation

The above script uses 'synchronous' serial communication. This means that the match function stops the script, effectively blocking the control panel until the response is received. As explained before, blocking the control panel is generally not a good idea. A better way to do this is to define a callback function for receiving the data:

```
s.onData = function(v){ label = v; };
```

Now the line:

```
s.match("MV?\r","\r",250);
```

will not block the control panel anymore. The script will finish, and when the response with the volume is received from the A/V receiver, the anonymous inline function is called, which will set the label.

You can also define callback functions for handling the timeout and other errors. The following lines make sure a diagnostics message is logged when a timeout or another error occurs:

```
s.onTimeout = function(v){ Diagnostics.log("A/V receiver timeout"); }
s.onError = function(e){ Diagnostics.log("A/V receiver error " + e); }
```

8.3. Inputs

The power sense inputs of the extender are equally easy to operate. To get the first power sense input of extender 0, just write:

```
var i = CF.extender[0].input[0];
```

Again, `i` will be `null` if the extender is defined as a basic extender.

8.3.1. Getting the state

Now, imagine you would want a panel on the page that should indicate the power state of a device. This can be done by tagging it "POWER_STATE" and adding a small page script to inquire the state of the input:

```
var i,w;
i = CF.extender[0].input[0];
w = widget("POWER_STATE");
w.label = i.get() ? "high" : "low";
```

This requests the state of the input from the extender and then updates the panel with the text "**high**" or "**low**" accordingly. When you configure the page script to be repeated, you will see the panel being updated when the input changes.

8.4. Relays

An extender relay port can be controlled as follows. First the corresponding Relay object is obtained:

```
var r = CF.extender[0].relay[0];
```

Then the current state can be retrieved with `get()` and changed with `set()` or `toggle()`:

```
if( r.get() == false ) {
    r.set(true);
}
```

8.5. Limitations

When using the extenders you should be aware of the fact that one extender can do only one thing at a time. So for example, while you are doing a receive operation on one serial port, you cannot ask it to send something on another port or toggle a relay etc. Also if you are implementing an installation with multiple control panels, you will get an error if you try to access a port of an extender that is currently processing a request for another control panel.

So try to write scripts that do not block the extenders for a long time. Suppose that your A/V receiver sends serial data when its volume is changed and that you want to reference these 'unsolicited events' to update the screen of the control panel accordingly. You could use the following script:

```
function PollAVReceiver(d)
{
    .../* parse d for data to be displayed */
    s.match("","\r",1000); // Collect data for one second
}
s.onData = PollAVReceiver;
PollAVReceiver(""); // Start polling
```

This will constantly read from the serial port and parse the received data to update the screen. But it will also keep the extender locked continuously. Instead, you could also write:

```
function PollAVReceiver()
{
    d = s.match("","\r",0); // Synchronous read with timeout=0
    .../* parse d for data to be displayed */
    scheduleAfter(1000, PollAVReceiver); // Schedule next poll
};
PollAVReceiver (); // Start polling
```

Or simply put this in the page script with a repeat interval of one second:

```
d = s.match("","\r",0); // Synchronous read with timeout=0
.../* parse d for data to be displayed */
```

This is a better solution since now the extender will only be locked for a very short time every second.

Chapter 9. Network connections

Another powerful feature of the Pronto is its ability to perform network communication via WiFi or Ethernet. The ProntoScript programmer can make use of this feature to interface with other IP networked devices. A network connection can be established using the `TCPSocket` class.

The following line creates a variable of type `TCPSocket` :

```
var socket = new TCPSocket(true);
```

Similarly to serial communication, network sockets can be used in a synchronous or asynchronous way. The parameter `true` above indicates synchronous, which means that the script will block during every socket operation, while in the asynchronous case callback functions are called at the completion of each operation.

9.1. Synchronous operation

The first thing to do when setting up a network connection is to specify the destination:

```
socket.connect('google.com', 80, 3000);
```

This call tries to connect to the website "`google.com`", port 80.

Instead of the name, also the ip address can be given, for example: `"192.168.42.110"`. When the destination is found within three seconds, the script continues, otherwise an exception will be thrown. See section Section 2.5, “Exceptions” on handling exceptions, but let's first describe the case that everything goes well.

Once the connection is established, it can be read from and written to. The following lines ask for the root directory using the HTTP protocol. Then it stores the first 100 characters that are received during maximally 3 seconds.

```
socket.write("GET / HTTP/1.0\r\n\r\n");
result = socket.read(100, 3000);
```

When finished, the connection should be closed:

```
socket.close();
```

The above code snippets can be combined in a single button script to show the result on the button label when it is pressed:

Example 9.1. Synchronous HTTP client

```
var socket = new TCPSocket(true);
socket.connect('google.com', 80, 3000);
socket.write("GET / HTTP/1.0\r\n\r\n");
label = socket.read(100, 3000);
socket.close();
```

Tip

It is recommended to close the connection after use.

It is possible to leave the connection open but you should always check for the status of the connection, by sending something over the TCP socket and reading back a response on that sent data within a certain timespan (assuming that the protocol used by the server supports something like this).

Even if one can write to a TCP socket, this does not always mean that the data will be immediately received by the other end. Only reception of new inbound data is a valid indication that the connection is still alive.

For example, the connection could have been closed by the remote end while the control panel was asleep (and thus unaware of this). At wake up, the control panel would still assume that the connection is open and a write operation to the socket will succeed. Eventually, this write will trigger an I/O error or an `onClose` event.

Of course you should make sure you properly configured the network settings for the control panel in the editor. Then, you can download this configuration to your control panel and test it.

When you press the button, you will notice that the script execution blocks the control panel while setting up the connection and getting the data. The next section shows how to avoid this.

Important

It is always preferred to use asynchronous communication because this will not block the user interface.

Synchronous communication will block the user interface until the timeout expired or until a match is received. Also, with asynchronous communication, it is less likely that the communication buffers get exhausted.

9.2. Asynchronous operation.

When specifying `false` when constructing the TCPSocket, an asynchronous socket is created:

```
var socket = new TCPSocket(false);
```

The next line looks identical to the synchronous case:

```
socket.connect('google.com', 80, 3000);
```

But now the `connect()` will return immediately and the script continues, although the connection is not yet established. Therefore writing to the socket is not possible yet. So, the remainder of the script should be executed when the connection is ready: in the `onConnect` callback function:

```
socket.onConnect = function()
{
    write("GET / HTTP/1.0\r\n\r\n");
};
```

So when the connection is established, the `onConnect` function is called which writes the request to the socket. Note that within this socket function, we can call the `write()` function without prefixing it with `socket`, because the socket scope is active. Refer to Section 6.2, “Scope” on scoping rules.

Then we want to read the response. But we cannot start reading yet, because no data is available yet and we do not want to block the control panel to wait for data. This is triggered by the `onData` callback function:

```
result = "";
socket.onData = function()
{
    result += read();
};
```

When data is available, the `onData` callback is triggered. This function can be triggered repeatedly, as long as data is coming in. That's why the above example accumulates everything in the `result` variable. Note that no count and no timeout are specified for the `read` function. It will return immediately with all available data.

Tip

When using asynchroneous `TCPSocket` operation, it's a good idea to always set an `onData` callback, and perform a `read` on the `TCPSocket` instance, even if the received data is not being used.

This will avoid communication buffer overflows.

According to the HTTP standard, the destination will close the socket when the document is completely transferred. This will trigger the `onClose` callback function that can show the accumulated result in the button label:

```
socket.onClose = function()
{
    label = result;
};
```

The combined script looks as follows:

Example 9.2. Asynchronous HTTP client

```
var socket, result;
socket = new TCPSocket(false);
result = "";
socket.onConnect = function()
{
    write("GET / HTTP/1.0\r\n\r\n");
};
socket.onData = function()
{
    result += read();
};
socket.onClose = function()
{
    label = result;
};
socket.connect('google.com', 80, 3000);
```

It is a little more extensive than the synchronous case, but it does not block the control panel.

One more thing that should be added is some error handling. In case of an error during one of the socket operations, the `onIOError` callback function is called, if defined:

```
socket.onIOError = function(e)
{
    label = "Socket error: " + e;
};
```

9.3. Reusing `TCPSocket` instances

Reusing a `TCPSocket` is not supported. Always create a new socket after a close:

```
function reInitSocket()
{
   socket = new TCPSocket();
   socket.onConnect = my_onConnect;
   socket.onData = my_onData;
   socket.onClose = my_onClose;
   socket.onIOError = my_onIOError;
   socket.connect(ipAddress, port);
}

  ...

if (socket.connected === true) {
   socket.write( ... );
} else {
   reInitSocket();
}
```

Chapter 10. Getting external images

In the previous chapter you have seen how you can set up a connection to a web server. This chapter will illustrate how to use this to got images from a web server to a Pronto.

By being able to dynamically create images, it is not necessary to include all images in the configuration file. Just set up a TCP connection to a web server and get your favorite images. And don't worry about the dimensions of the images! With the `stretchImage` property, all images in a widget are stretched to fit the widget size.

Before showing the complete code it could be interesting to explain a few lines. The most important feature is of course the dynamic creation of images. You can make your own images by calling the `Image` class constructor.

```
var MyImage = new Image(bitmapData);
```

The bitmapData is raw image data that is in a PNG, JPG or BMP format and it is stored as a `String`. Normally you get this data after opening a TCP connection to a web server and downloading the image data, but you can even create it manually in the Pronto.

The second line of code that needs a little clarification is the following:

```
widget("output").stretchImage = true;
```

By executing this line of code just once, all images placed in the `output` widget will be stretched to fit the widget size.

A complete example is below:

Example 10.1. Displaying an image from an HTTP server

```
var socket, receivedData;
socket = new TCPSocket();
receivedData = "";

socket.onConnect = function() {
        write("GET /images/img1.jpg HTTP/1.0\r\n\r\n");
};

socket.onData = function() {
        receivedData += read();
};

socket.onIOError = function (e) {
        widget("output").label = "IOError " + e;
};

socket.onClose = function () {
        var imageStartIndex,
            bitmapData,
            myImage;

        // remove the HTTP header from the received data
        imageStartIndex = receivedData.indexOf("\r\n\r\n");
        bitmapData = receivedData.substring(imageStartIndex+4);

        // make and display the image
        myImage = new Image(bitmapData);
        widget("output").setImage(myImage);
};

socket.connect("MyServer.com",80,3000);
```

Tip

When using the `com.philips.HttpLibrary` library, the above example can be replaced with:

```
System.include("com.philips.HttpLibrary.js");
var httpLib = com.philips.HttpLibrary;
httpLib.showHTTPImage('http://MyServer.com/images/img1.jpg',
                      'output');
```

(Also see Section 11.1, “Using a library”)

As can been seen, most of the code is needed to set up the connection to the server. When all the data is received, the HTTP header information is stripped, the image data is extracted, and rendered to a widget.

Chapter 11. Libraries

A ProntoScript Library is a self-contained file which allows a script to be easily reused in multiple activities, pages and XCFs.

11.1. Using a library

When you want to use an existing library in your XCF, you drag & drop it from the Building Blocks on an activity in the Project Overview or on the PS Libraries tab of the Activity Properties.

This will associate the library with the activity and will cause it to be read whenever the project is being saved, simulated or downloaded to the control panel.

Note

By default the ProntoScript Libraries tab is not shown in the Building Blocks.

If you want to use the ProntoScript Libraries, select the Options... menu entry of the Tools menu in the main menu bar, to show the Options dialog. In that dialog, select the General Settings tab, and select Advanced view in the View Mode Selection box.

After the library is associated, it needs to be included. This is done with the ProntoScript System.include method.

```
System.include("com.example.MyLibrary.js");
```

Tip

The filename of a library can be copied to the clipboard by using the **Ctrl+C** shortcut while a library in the PS Libraries tab of the Activity Properties is selected.

When distributing the XCF, the libraries in the XCF will be copied to the user's library folder.

11.1.1. System-wide libraries

If a library is required in multiple activities, ProntoEdit Professional allows adding this library globally, in the PS Libraries tab of the System Properties pane. This will cause the library to be attached to each activity in the project.

11.2. Installing a library

Libraries can be added manually, by storing them in the `Libraries` folder of your installation.

Note

Depending on the Windows version, libraries are stored at the following location:

Windows XP	`C:\Documents and Settings\All Users\Application Data\Philips\ProntoEdit Professional 2\Libraries`
Windows Vista	`C:\ProgramData\Philips\ProntoEdit Professional 2\Libraries`

11.2.1. Version Control

When an XCF is opened in the editor, the contained libraries are validated. After the libraries are validated, the following conditions are checked:

library is invalid	library will be validated	
library is not already installed	library is copied to disk	
library is already installed	local library has higher version	library in XCF is backed up, local version will be used
	local library has lower version	local library is backed up and will be replaced by library in XCF

11.3. Creating a library

There are 2 types of libraries: protected and unprotected libraries. Protected libraries are binary files which have a `.pjs` extension, while unprotected libraries are plain text files with a `.js` extension.

This section covers the creation of unprotected libraries.

Unprotected libraries are JavaScript files (in UTF-8 encoding), which can be created using any text editor.

For an unprotected library to be recognized by ProntoEdit Professional, it must contain a special comment block (delimited with `/*!` and `*/`) within the first 100 lines of the file. This header should contains three mandatory fields. The order of these fields is free, but they must appear on different lines. Other lines can be freely added in the header, they will not be parsed.

`@author` minimum 1, maximum 80 of the following characters: [a-zA-Z0-9 ,.;-_:/@]

`@title` minimum 1, maximum 80 of the following characters: [a-zA-Z0-9 ,.;-_:/@]

`@version` format *`<number>.<number>`*, where *`number`* is a number between 0 and 999. This can be followed by whitespace, after which the rest of the line is ignored.

Example 11.1. Library header

```
/*!
  @author Koninklijke Philips Electronics
  @title com.philips.CurrencyConverter
  @version 1.0
 */
```

It is recommended to use the *Java Package Naming Convention* for library filenames and titles. This convention states that the file should have a hierarchical name with the following parts:

- the top level domain name of the organization
- the organization's domain
- any subdomains listed in reverse order
- the name of the library

11.4. Protecting libraries

A protected library is a library which is cryptographically protected to prevent easy retrieval of the source code, and to protect against tampering. When a protected library is modified even slightly, the control panel will be refuse to use it.

To protect a library, right click on the library in the Building Blocks and select Protect Library from the popup menu.

Note

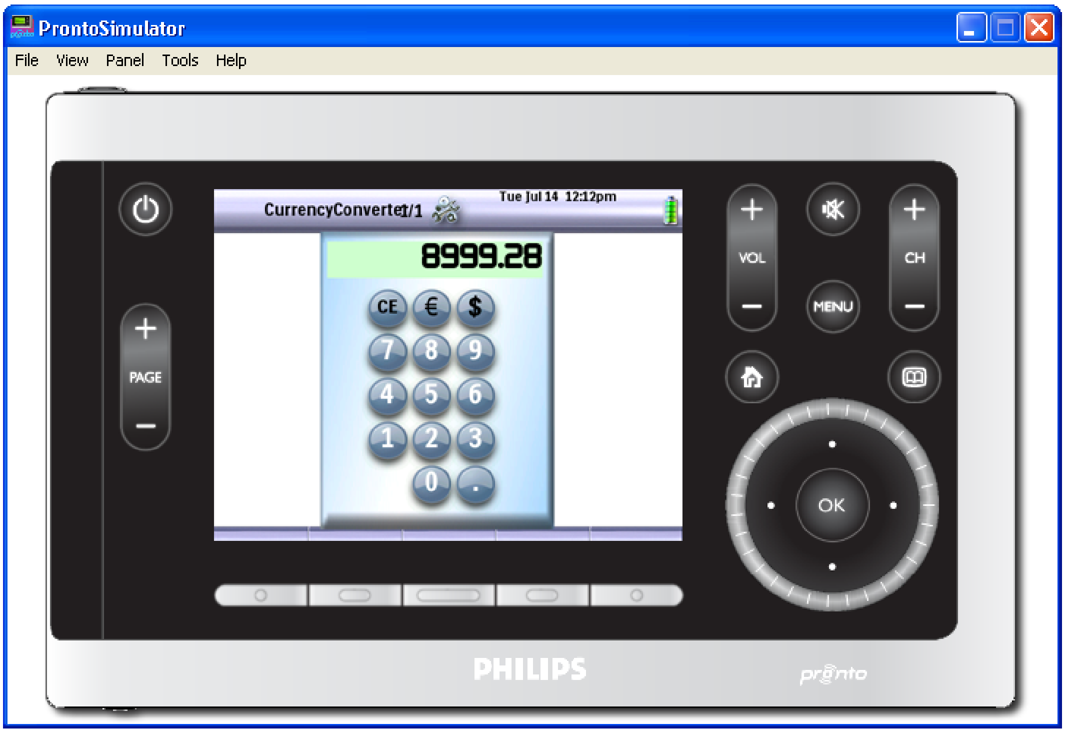
A USB connection with a control panel is required to be able to protect a library.

Note

Protected scripts can only be executed on the control panel, and are not available in the simulator.

11.5. Library example: Currency Converter

This section describes a library implementing a basic currency convertor, in which almost all of the functionality is contained within a library. Activity, page and button scripts are only used to hook up the GUI with the library implementation.



```
/*!
  @title com.philips.CurrencyConverter
  @version 1.0
  @author Koninklijke Philips Electronics
 */

// Setup com.philips.CurrencyConverter namespace
var com;
if (!com) {
  com = {};
} else if (typeof com !== "object") {
  throw new Error("com already exists and is not an object");
}
```

```
if (!com.philips) {
  com.philips = {};
} else if (typeof com.philips !== "object") {
  throw new Error("com.philips already exists and is not an object");
}
if (com.philips.CurrencyConverter) {
  throw new Error("com.philips.CurrencyConverter already exists");
}
com.philips.CurrencyConverter = {};

// Define and invoke anonymous function to fill
// private namespace.
(function () {

  var exchangeRate = 1.389,
      gotDot = false,
      displayWidget,
      ns;

  function updateDisplay(text)
  {
    // Once an error happened, no display updates are done anymore
    // until cleared
    if (displayWidget.label !== 'ERROR') {
      if ((displayWidget.label === '0') || (text === 'ERROR')) {
        displayWidget.label = '';
      }
      displayWidget.label += text;
    }
  }

  function clearEverything()
  {
    displayWidget.label = '0';
    gotDot = false;
  }

  function processDigit(digitText)
  {
    updateDisplay(digitText);
  }

  function processDot()
  {
    if (gotDot === false) {
      updateDisplay('.');
      gotDot = true;
    }
  }

  function reportError()
  {
    updateDisplay('ERROR');
  }

  function convert(toCurrency)
  {
    var convertedValue;

    switch (toCurrency) {
    case 'Euro':
      convertedValue = parseFloat(displayWidget.label, 10) / exchangeRate;
      convertedValue = convertedValue.toFixed(2);
      displayWidget.label = convertedValue;
      break;
    default:
      break;
    }
  }

  function setDisplayWidget(widget)
  {
    displayWidget = widget;
    clearEverything();
  }

  function process(value)
  {
    if (displayWidget) {
      switch (value) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
        processDigit(value);
        break;
      case '.':
        processDot();
        break;
      case 'CE':
        clearEverything();
        break;
      case 'toEuro':
        convert('Euro');
        break;
      default:
        reportError();
        break;
      }
    }
  }

  // Export public symbols, other symbols will remain
  // private.
  ns = com.philips.CurrencyConverter;
  ns.process = process;
  ns.setDisplayWidget = setDisplayWidget;
}());
```

The above library must be added to an activity, in the PS Library tab of the Activity Properties pane.

This library should only be executed once in the activity (to define and populate the namespace), so it's best to include it in the activity script.

Example 11.2. Using CurrencyConverter: activity script

```
System.include("com.philips.CurrencyConverter.js");
```

A page script can be used to set the panel used for rendering the conversion result.

Example 11.3. Using CurrencyConverter: page script

```
var currencyConverter = com.philips.CurrencyConverter;
currencyConverter.setDisplayWidget(GUI.widget('display'));
```

Finally, each of the buttons can be fitted with a small button script, which invokes the `process` method of the `com.philips.CurrencyConverter` object. (Another approach would be to use ProntoScript tags to hook up buttons to the library)

Example 11.4. Using CurrencyConverter: Euro button script

```
currencyConverter.process("toEuro");
```

Example 11.5. Using CurrencyConverter: Digit button script

```
currencyConverter.process("7");
```

Chapter 12. Creating ProntoScript Modules

A ProntoScript module is an XCF file with one activity containing a number of pages and scripts that control a specific device. This activity should be self-contained. This means that the scripts should not refer to widgets in other activities or on the system page.

12.1. Merging XCFs to import activities

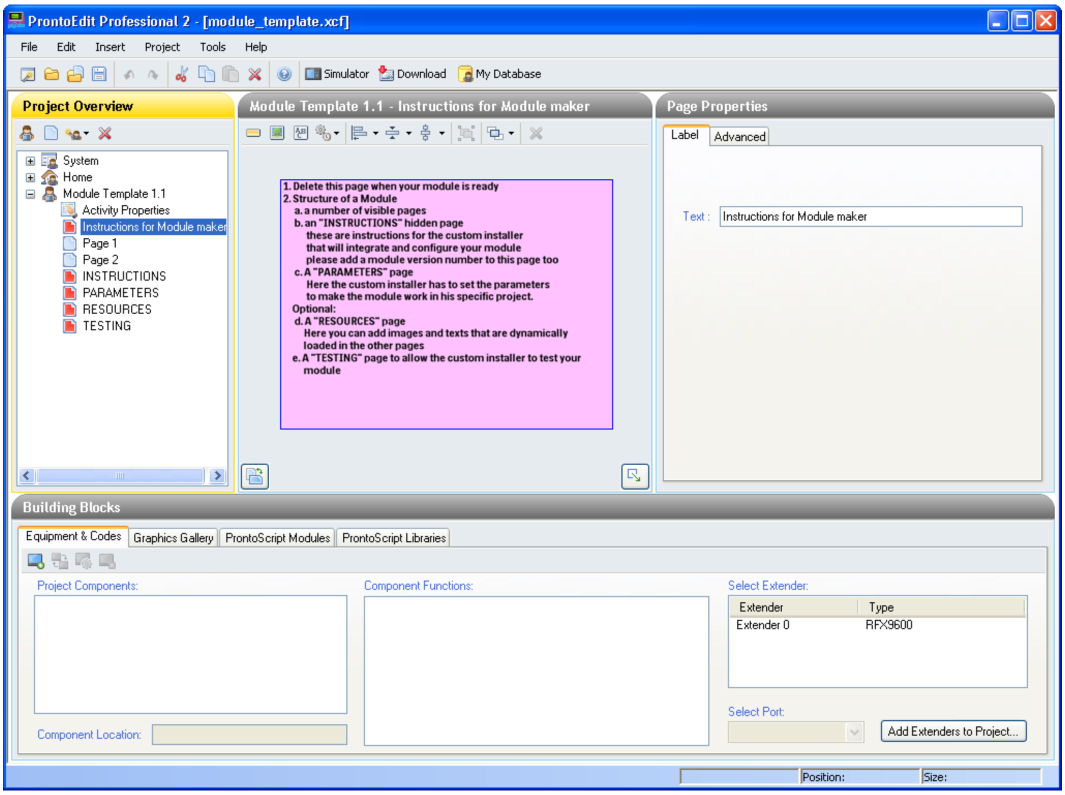
When you want to use the specific device, you can include its activity in your configuration file by merging it with the ProntoScript module via the "Merge Project" menu item in the editor. This will add the activity at the bottom of the activity list.

12.2. Using hidden pages for easy configuration

Then the module probably needs to be configured. For example which extender should be used to control the device, what ports are connected, buffer sizes, error levels, etc. The first hidden page should be called "`INSTRUCTIONS`" and should contain help information to the custom installer.

Next is a page "`PARAMETERS`" with yellow fields (panels) which configure the Pronto to communicate properly to the device to be controlled. These pages should be made hidden, so that they are only visible in the editor and not on the Pronto to the end user.

In an effort to standardize the custom installer experience with ProntoScript modules, the following template is proposed. It is strongly encouraged to make use of it.



1. Delete this page when your module is ready
2. Structure of a Module
   a. a number of visible pages
   b. an "INSTRUCTIONS" hidden page
      these are instructions for the custom installer
      that will integrate and configure your module
      please add a module version number to this page too
   c. A "PARAMETERS" page
      Here the custom installer has to set the parameters
      to make the module work in his specific project.
   Optional:
   d. A "RESOURCES" page
      Here you can add images and texts that are dynamically
      loaded in the other pages
   e. A "TESTING" page to allow the custom installer to test your
      module

Text : Instructions for Module maker

Extender	Type
Extender 0	RFX9600

Note

ProntoScript scoping rules allow modules to be included more than once in a single project without interfering with each other. An exception (by definition) is the use of Global variables. Therefore it is advised to prefix the Global variable names with the Module name.

The installer will typically change this, making the string unique. You can of course ask him to do so explicitly in the INSTRUCTIONS page.

The parameters can be retrieved from the PARAMETERS page and stored in variables in the activity script when the module is started. For example:

```
var server_url = CF.widget("PARAMETER1", "PARAMETERS").label;
var ip_address = CF.widget("PARAMETER2", "PARAMETERS").label;
var port_nr    = CF.widget("PARAMETER3", "PARAMETERS").label;
...
```

If you need more than four parameters, add another parameter page with the same layout. Make sure the tags and labels of the parameters on this new page are numbered correctly.

Chapter 13. Exceptional Scenarios

13.1. Out of memory

When a script runs out of memory, the script engine tries to free up memory with a process called 'garbage collecting'. This reorganizes the memory space allocated to the script engine in order to recover chunks of memory that are not used anymore. If this process does not free up enough memory, script execution will be halted and a diagnostic message will be logged. When the garbage collection process takes more than one second, also a diagnostic message will be logged.

13.2. Nested scripting

Nested scripting is prohibited. When a script is triggered while the script engine is already executing another script, it will be queued after the engine is finished. This also means that event functions will be called after the current script is finished.

13.3. Infinite scripts

It is possible to create a script that takes a long time to execute and effectively blocks the control panel. In order to enable the user to fix this situation, a key combination can be pressed during the start-up of the control panel that disables the script engine. The key combination to be used is: **Backlight+Menu+ChannelUp**. It must be pressed continuously during the start-up animation and the please wait screen. A diagnostic message will be logged to indicate the limited functionality available. The user can then use the normal download procedure to download a corrected configuration file into the control panel. Another reboot is required to start the script engine again.

13.4. Invalid arguments

When an invalid value is set to a class property, or when a class function is called with invalid or insufficient parameters, a diagnostic message will be logged and the execution of the erroneous script will be stopped.

13.5. Script Exceptions

When an abnormal situation is detected during script execution, a script exception is generated. This can be any of the following:

Exception	Description
"Failed"	The operation failed, e.g. reading from an extender serial port timed out.
"Not Implemented"	A class property or method was used that is currently not implemented.
"Not Available"	A class property or method was used that is not available.
"Insufficient internal memory available"	Not enough memory when reading from a TCP socket or setting a global variable.
"Invalid name"	The name passed to `System.getGlobal()` or `setGlobal()` is not a proper string.
"Expected a function"	The specified callback is not a function.

Exception	Description
"Expected an integer"	The parameter passed is not an integer.
"Expected a positive integer"	The specified `Page.repeatInterval` is negative.
"No argument specified"	Not enough arguments are passed to the class method.
"Not enough arguments specified"	Not enough arguments are passed to the class method.
"Argument is not a string"	`TCPSocket.connect` expects a string as host name.
"Argument is not a function"	`Activity.scheduleAfter` expects a function.
"Argument is not an integer number"	A timeout must be an integer number.
"Argument is not a positive integer number"	A timeout or the number of bytes to receive must be larger than 0.
"Argument is not an image"	`Widget.setImage()` is called with an invalid argument.
"Argument is not a boolean"	`Input.match()` is called with an invalid argument.
"Limit of simultaneous timers reached"	The maximum number of pending `scheduleAfter` invocations has been reached.
"Socket error"	A socket operation resulted in an error. E.g. a `read()` or `write()` failed.
"Maximum active socket count reached"	The maximum number of sockets are already in use.
"Failed to connect"	`TCPSocket.connect()` failed. Check your network settings.
"Maximum blocking read length exceeded"	You tried to read more than 65536 bytes from a synchronous socket.
"Maximum read length exceeded"	You tried to read more than 512 bytes from a serial port.
"ActionList Error"	Error during `executeActions()`.

Chapter 14. Debugging your script

There are a number of ways to help you debug your script in case it does not work as expected. Or it does not work at all because of a typing mistake.

14.1. Debug widget

The first thing you can do is to create a debug widget on the page you want to debug. Create a text field with the tag _PS_DEBUG_.

In the Appearance tab, check the position it in the upper left corner and resize it to full screen.

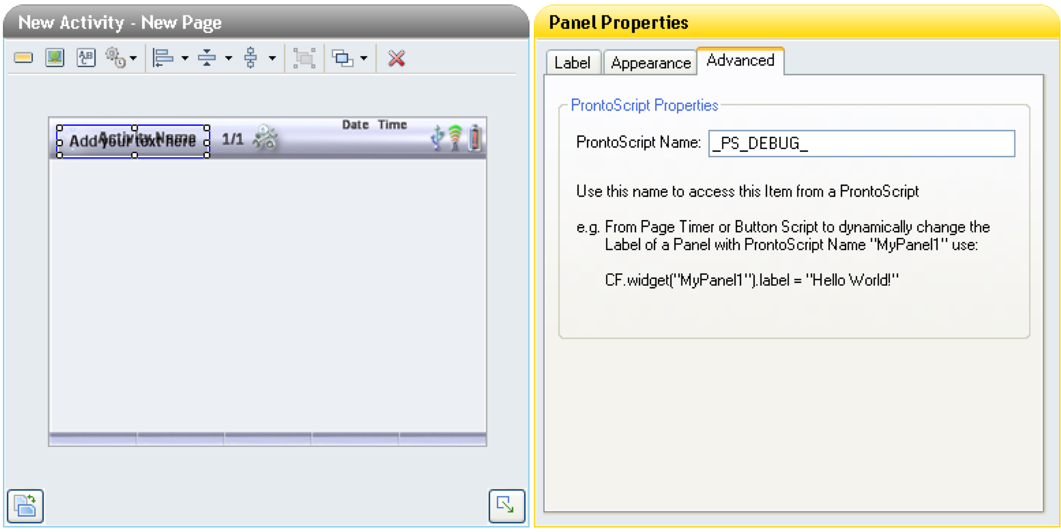
Then, in the Label tab, set the text alignment to bottom left.

Note

When using ProntoEdit Professional 1.x, this can be accomplished by creating a panel with the tag _PS_DEBUG_. In the Dimensions tab, position it in the upper left corner and resize it to full screen.

In the Appearance tab, check the No Fill box to make it transparent so you can still see and touch the other widgets on the page.

Then, in the Label tab, set the text alignment to bottom left.

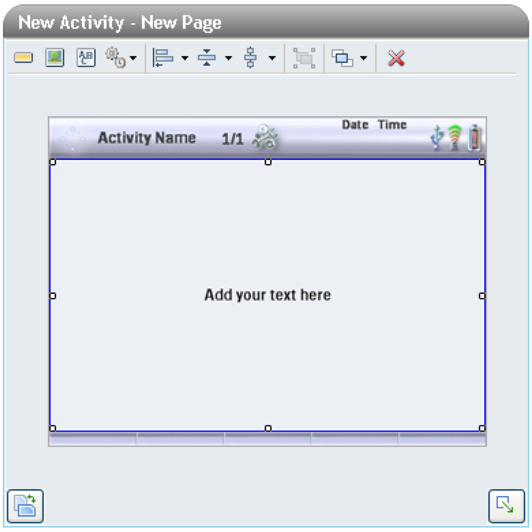


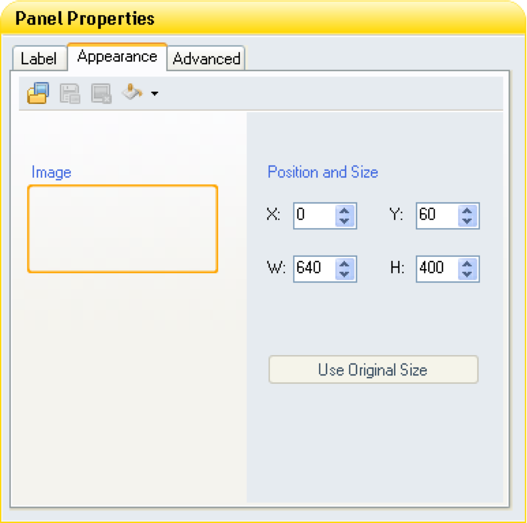
ProntoScript Name: _PS_DEBUG_

Use this name to access this Item from a ProntoScript

e.g. From Page Timer or Button Script to dynamically change the Label of a Panel with ProntoScript Name "MyPanel1" use:

```
CF.widget("MyPanel1").label = "Hello World!"
```

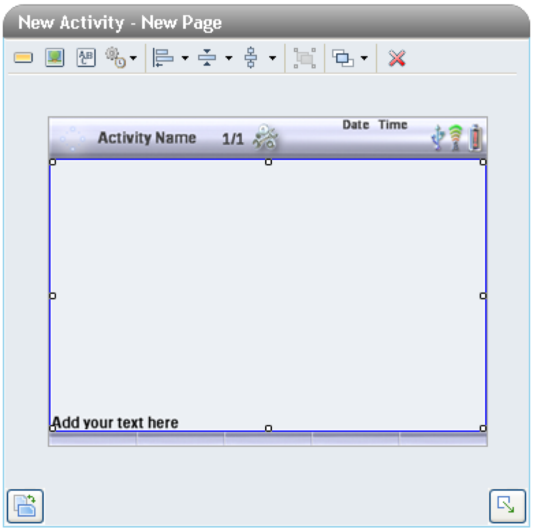


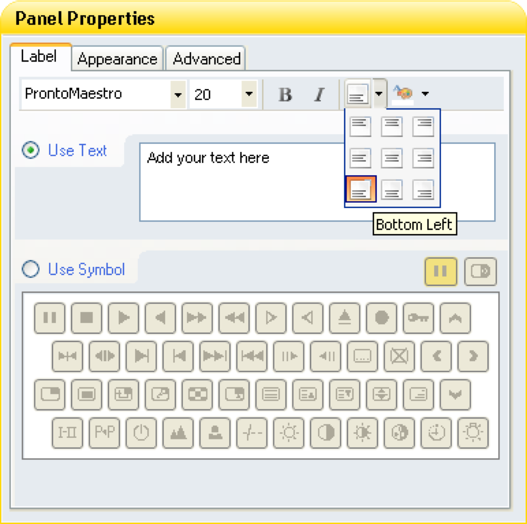


Position and Size

X: 0 Y: 60

W: 640 H: 400





ProntoMaestro 20

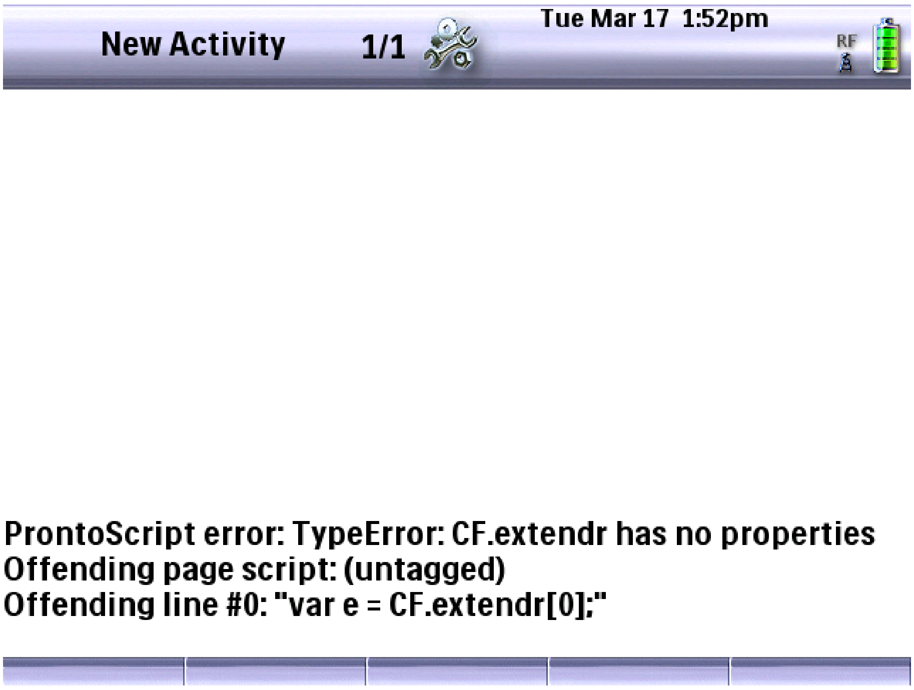
● Use Text: Add your text here

Now, if an error occurs when compiling or executing your script, an appropriate error message will be logged into this debug widget. It will indicate the offending script, the line number and a short description of the error.

Suppose that is a typo in a page script:

```
var e = CF.extendr[0];
```

This will give the following output on the screen:



ProntoScript error: TypeError: CF.extendr has no properties
Offending page script: (untagged)
Offending line #0: "var e = CF.extendr[0];"

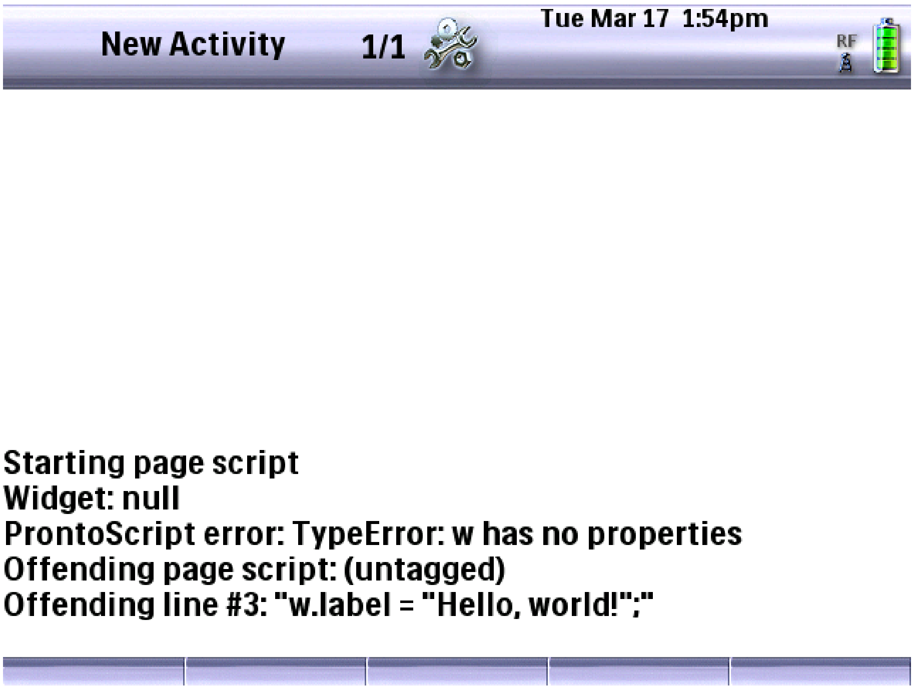
14.2. System.print()

You can add messages yourself to the debug widget while the script is running. This is done with the `System.print()` function. An example:

Example 14.1. `System.print()`

```
System.print("Starting page script");
var w = widget("WRONG_TAG");
System.print("Widget: " + w);
w.label = "Hello, world!";
System.print("Page script finished");
```

This will give the following output:



```
Starting page script
Widget: null
ProntoScript error: TypeError: w has no properties
Offending page script: (untagged)
Offending line #3: "w.label = "Hello, world!";"
```

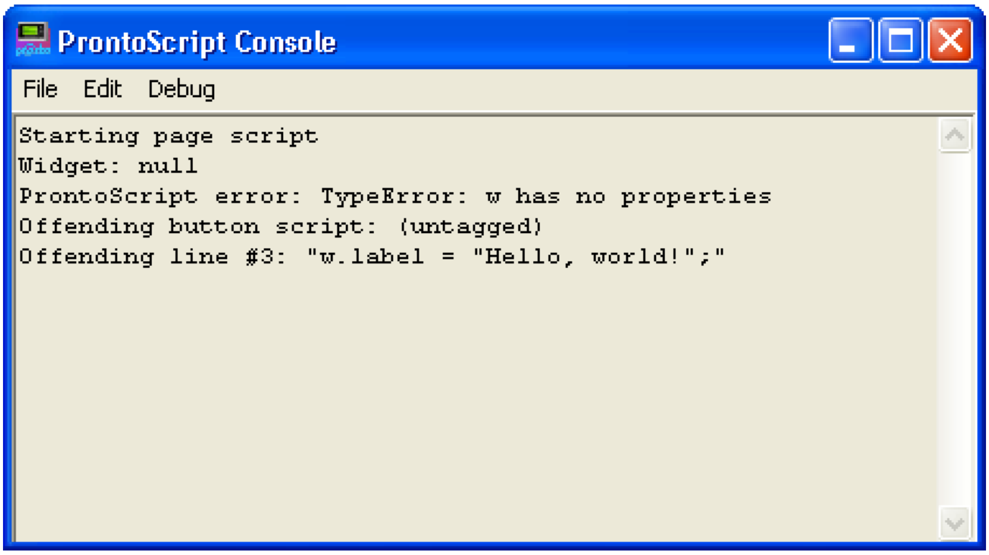
Note

You can pass any string to the `System.print()` function, but the text will be truncated to 99 characters.

14.3. ProntoScript Console

An alternative to the _PS_DEBUG_ widget is to use to ProntoScript Console available in the simulator.

All output which on the control panel would be written to a _PS_DEBUG_ widget (uncaught exceptions and `System.print()` invocations) can bee seen in the ProntoScript Console:

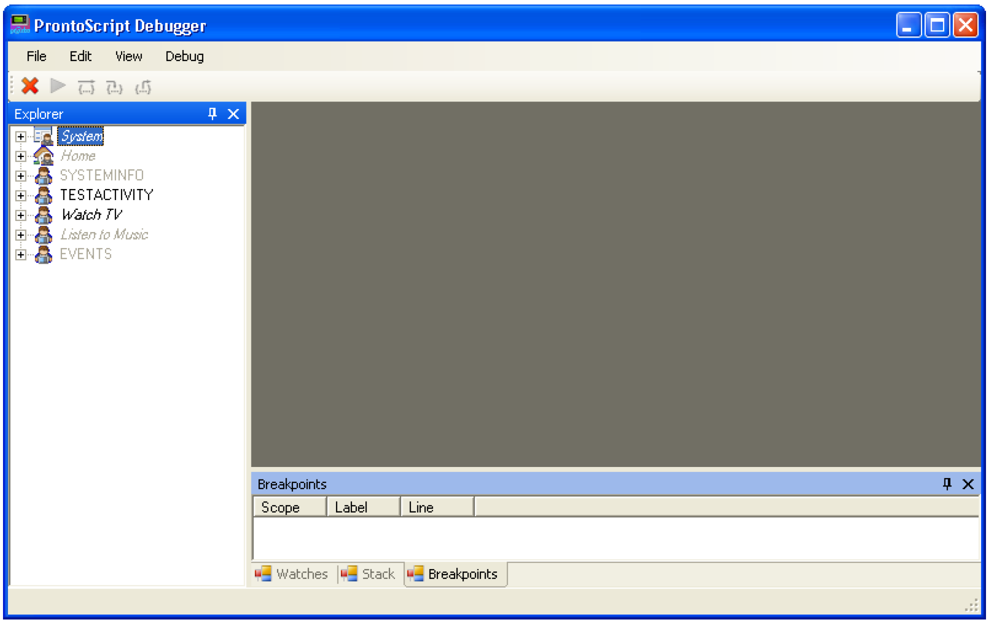


```
Starting page script
Widget: null
ProntoScript error: TypeError: w has no properties
Offending button script: (untagged)
Offending line #3: "w.label = "Hello, world!";"
```

14.4. ProntoScript Debugger

For more advanced script debugging, an interactive debugger is also available in the simulator. This allows tracing script execution, setting breakpoints and tracking variables.

To launch the debugger, select the ProntoScript Debugger menu entry of the Tools menu in the Pronto Simulator window.



14.4.1. Toolbar

The debugger's toolbar contains buttons to control script execution within the simulator:

✖

Stop. Stops script execution in the simulator.

Note

Since scripts interact with other GUI elements in the control panel, all GUI event processing will also be halted.

▶

Continue. Resumes script execution in the simulator.

{...}

Step Over. Resumes script execution, until a different source code line is reached, within the current closure. In most cases, this means that called functions will not be traced.

Note

Scripts are first compiled to optimized bytecode before they are executed. In some cases, this causes the order in which lines get executed seem unlogical.

For example, when a variable is declared in a given line, that bytecode corresponding to that line might not be executed until that variable is actually used.

{...} **Step In.** Resumes script execution, until a different source code line is reached (in any script).

{...} **Step Out.** Resumes script execution, until the end of the current closure. In most cases, this means that execution is resumed until the currently executing function is left.

14.4.2. Explorer Window

The explorer window provides a means to navigate through the elements in configuration which can be accessed from ProntoScript. These include all activities, pages and libraries, as well as those widgets which have a script or a ProntoScript tag.

Configuration elements which do not have a script attached as shown with a greyed out name. If they do have a script attached, double clicking on them will open a window with the corresponding script source.

Configuration elements which do not have a ProntoScript tag are shown in italics, and are shown with their label.

14.4.3. Script Windows

Script windows provide a read-only view of a script's source.

If execution is halted by the debugger, the currently active source code line will be shown with a yellow background.

Breakpoints can be set and cleared by clicking in the left-hand area, before the line number. (A red bullet will be shown for lines on which a breakpoint is currently set)

14.4.4. Breakpoints Window

The Breakpoints window lists the currently active breakpoints. Double-clicking on a breakpoint in this list will cause the breakpoint line to be shown in a script window. Right-clicking a breakpoint entry will show a context menu which can be used to remove the breakpoint.

14.4.5. Watches Window

The Watches window shows a list of expressions which will be evaluated every time the debugger halts execution of the simulator. Watch expressions can be added or removed by right-clicking in the Watches window.

14.4.6. Stack View

The Stack View window shows the chain of invoked functions up to the point where script execution was last halted.

Appendix A. ProntoScript Classes Description (ProntoScript API)

The Maestro control panel scripting language provides a number of object classes that can be accessed and provide access to the internals of the control panel. The following sections list the available script object classes in alphabetical order.

A.1. Activity class

This class represents a control panel activity as defined in the editor. An activity is in fact a collection of pages with common hard key definitions.

Instance Properties

`label`	This is the text that is shown in the ActivityName status field when a page of the activity is displayed.
`tag`	The tag is the activity name within the script. It is used to find a specific activity in the configuration file.
`onRotary`	Define the function to be executed after a rotation of the rotary.
`onSleep`	Define the function to be executed when the panel is about to enter standby (sleep) mode.
`onWake`	Define the function to be executed when the panel is woken up from standby (sleep) mode.
`wifiEnabled`	Allows restarting, enabling and disabling the network interface, if it is enabled in the configuration file.

Class methods

`scheduleAfter()`	Program a function to be executed once after a certain time.

Instance methods

`page()`	Find the page with tag tagP in the activity.
`widget()`	Search the activity for a widget in a specific page.

A.1.1. Instance properties

A.1.1.1. Activity.label

Purpose

This is the text that is shown in the ActivityName status field when a page of the activity is displayed.

Read/Write : RW

The label of all activites can be changed, but the change will only persist as long as the current activity is active.

Value : String

The text can be of any length but the number of characters displayed will depend on the size of the activity status field widget.

Additional info

The activity name status widget displays the label of the current activity. This is initially the name that is defined in the configuration file. By writing a value to the label property the displayed activity name can be changed. When `null` is written to the label, the original name is restored.

Example

Example A.1. Activity.label

```
CF.activity().label = "Busy..."; // Show Busy instead of the activity name
CF.activity().label = null; // Show activity name again
```

A.1.1.2. Activity.tag

Purpose

The tag is the activity name within the script. It is used to find a specific activity in the configuration file.

Read/Write : R

Value : String

When no tag is defined an empty string is returned

A.1.1.3. Activity.onRotary

Purpose

Define the function to be executed after a rotation of the rotary.

Read/Write : RW

Value : onRotaryCallback

The function to be called.

A.1.1.4. Activity.onSleep

Purpose

Define the function to be executed when the panel is about to enter standby (sleep) mode.

Read/Write : RW

Value : onSleepCallback

The function to be called.

A.1.1.5. Activity.onWake

Purpose

Define the function to be executed when the panel is woken up from standby (sleep) mode.

Read/Write : RW

Value : onWakeCallback

The function to be called.

A.1.1.6. Activity.wifiEnabled

Purpose

Allows restarting, enabling and disabling the network interface, if it is enabled in the configuration file.

Read/Write : RW

Value : Boolean

Reads out `true` if the network interface is enabled, `false` if the network interface is disabled.

When writing `true` to this property, the network interface will be restarted.

Setting this property to `false` will cause the network interface to be disabled (conserving battery power), until an activity switch is performed.

A.1.2. Class methods

A.1.2.1. Activity.scheduleAfter()

Synopsis

```
Activity.scheduleAfter(duration,onAfter)

Activity.scheduleAfter(duration,onAfter,id)
```

Purpose

Program a function to be executed once after a certain time.

Parameters

duration — `Integer`

The duration after which the function should be executed in milliseconds. Must be greater than 0.

onAfter — `Function`

The function to be scheduled.

id — `any`

	Optional parameter. The id can have any type and is passed as a parameter to the `onAfter` function to enable usage of a generic event function.

Return

`void`

Exceptions

- Not enough arguments specified
- Argument is not an integer
- Argument is not a function
- Limit of simultaneous timers reached

Additional info

The function is only called if the activity is still active after the specified duration. Multiple functions can be scheduled in parallel with different durations. The execution of the functions will be scheduled sequentially. A maximum number of 10 scheduled functions are supported in parallel.

Note

All control panel timers are paused while the control panel is asleep, postponing all pending function calls.

A.1.3. Callback functions

The prototypes of the call back functions are listed below. In the call back functions, you can use '`this`' to refer to the scope of the actual input object that is causing the call back.

A.1.3.1. *`onRotaryCallback`*

Purpose

Called after a rotation of the rotary.

Parameters

`clicks`	`Integer`

Additional info

The parameter is positive after a clockwise rotation, negative after an anticlockwise rotation.

When the user stops rotating the rotary wheel, the last value is always a 0.

Example

Example A.2. *`onRotaryCallback`*

```
onRotary = function(clicks)
{
    // put the rest of your code here
};
```

A.1.3.2. *onSleepCallback*

Purpose

Called when the panel is about to enter standby (sleep) mode.

Parameters

None.

Example

Example A.3. *onSleepCallback*

```
onSleep = function()
{
    // put the rest of your code here
};
```

A.1.3.3. *onWakeCallback*

Purpose

Called when the panel is woken up from standby (sleep) mode.

Parameters

None.

Example

Example A.4. *onWakeCallback*

```
onWake = function()
{
    // put the rest of your code here
};
```

A.1.4. Instance methods

A.1.4.1. Activity.page()

Synopsis

activity.page()

activity.page(*tagP*)

Purpose

Find the page with tag tagP in the activity.

Parameters

tagP	`String` The tag of the page to search for. May be empty, `null` or omitted.

May be a predefined tag.

Return

`Page`

The class instance corresponding to the found page, or `null` if no page found with the specified tag.

If no tag is specified, the current page is returned of the current activity.

Exceptions

- Argument is not a string

Additional info

See Section A.8, “Page class” for a description of the return value.

Refer to Appendix C, *Predefined tags* for the applicable predefined page tags.

A.1.4.2. Activity.widget()

Synopsis

```
activity.widget(tagW)
```

```
activity.widget(tagW, tagP)
```

Purpose

Search the activity for a widget in a specific page.

Parameters

`tagW`	`String` The tag of the widget to search for as defined in the editor.
`tagP`	`String` The tag of the page in which to search. May be empty, `null` or omitted. In that case the current page of the current activity is searched for the widget.

Return

`Widget`

The class instance corresponding to the found widget, or `null` if not found.

Exceptions

- Not enough arguments specified
- Argument is not a string

Additional info

Refer to Section A.13, “Widget class” for detailed information on the return type.

Refer to Appendix C, *Predefined tags* for the applicable x predefined page tags.

A.2. CF class

This class gives access to the configuration file of the control panel, containing all items programmed by the editor.

Class Properties

`extender`	Array that provides access to the extenders defined in the configuration file. The array has a fixed size of 16 elements. Each element matches the corresponding extender as configured in the editor.

Class methods

`activity()`	Provide access to one of the activities that are defined in the configuration file.
`page()`	Provide access to one of the pages in the configuration file.
`widget()`	Search the configuration file for a specific button, panel, hard key or firm key and returns the corresponding Widget class instance.

A.2.1. Class properties

A.2.1.1. CF.extender

Purpose

Array that provides access to the extenders defined in the configuration file. The array has a fixed size of 16 elements. Each element matches the corresponding extender as configured in the editor.

Read/Write : R

Value : Extender

An entry refers to a valid Extender class instance, or undefined if no extender with that id is defined in the configuration file. Note that `null` and undefined are equal in ProntoScript.

Additional info

Refer to Section A.4, “Extender class” for an extensive description of its properties.

Example

Example A.5. CF.extender

```
// Locates extender 0 and checks if it is configured:
var e = CF.extender[0];
if (e == null)
    Diagnostics.log("Extender 0 not available");
```

A.2.2. Class methods

A.2.2.1. CF.activity()

Synopsis

`CF.activity()`

`CF.activity(tagA)`

Purpose

Provide access to one of the activities that are defined in the configuration file.

Parameters

tagA	`String` The tag to look for. May also be empty, `null` or omitted.

Return

`Activity`

The first found activity object with the specified tag, or `null` if no activity was found in the configuration file with that tag. If no parameter is specified, or if an empty string is passed, the current activity object is returned.

Exceptions

- Argument is not a String

Additional info

Refer to Section A.1, “Activity class” for detailed information on the activity members.

Refer to Appendix C, *Predefined tags* for the tags to be used for the home and system activity.

Example

Example A.6. CF.activity()

`CF.activity("DVD")`: returns the activity tagged "DVD".

`CF.activity("")`: returns the current activity object.

A.2.2.2. CF.page()

Synopsis

`CF.page()`

`CF.page(tagP)`

`CF.page(tagP, tagA)`

Purpose

Provide access to one of the pages in the configuration file.

Parameters

tagP	`String`
	Tag name of the page to search for. If both *tagA* and *tagP* are omitted, empty or `null`, the current page is returned of the current activity. In this case *tagA* is ignored.
tagA	`String`
	Tag name of the activity in which to search. If omitted, empty or `null`, the current activity is searched.

Return

`Page`

A Page class instance corresponding to the first page found with the tag *tagP* in the activity with tag *tagA*.

Exceptions

- Argument is not a string

Additional info

Refer to Section A.8, “Page class” for detailed information on the page class members.

Refer to Appendix C, *Predefined tags* for the tags to be used for the home and system page.

Example

Example A.7. CF.page()

`CF.page("2", "DVD")`: returns the page with tag "2" from the activity tagged "DVD".

`CF.page("Macros")`: searches the current activity for the page tagged "`Macros`".

`CF.page();`: returns the current page.

A.2.2.3. CF.widget()

Synopsis

`CF.widget(`*tagW*`)`

`CF.widget(`*tagW*`,`*tagP*`)`

`CF.widget(`*tagW*`,`*tagP*`,`*tagA*`)`

Purpose

Search the configuration file for a specific button, panel, hard key or firm key and returns the corresponding Widget class instance.

Parameters

tagW	`String`

	The name of the widget to search for.
tagP	`String` Tag name of the page that contains the widget. If both *`tagA`* and *`tagP`* are omitted, empty or `null`, the current page is searched.
tagA	`String` Tag name of the activity that contains the widget. If *`tagA`* is omitted, empty or `null`, the current activity is searched.

Return

`Widget`

The class instance corresponding to the found widget, or `null` if not found.

Exceptions

- Not enough arguments specified
- Argument is not a string

Additional info

The search order is not defined. Therefore it is not advisable to give the same tag to multiple activities, pages or widgets.

Refer to Section A.13, "Widget class" for detailed information on the return type.

Refer to Appendix C, *Predefined tags* for the applicable predefined page tags.

Example

Example A.8. CF.widget()

`CF.widget("AllOn", "2", "DVD")`: searches the widget tagged "`AllOn`" on the page with tag "2" on the activity tagged "DVD".

`CF.widget("On", "Macros")`: returns the button tagged "`On`" on the "`Macros`" page in the current activity.

A.3. Diagnostics class

The diagnostics class is be used to log messages in the diagnostics list. This list can be inspected by pressing and holding the following buttons in the stated order: **Backlight+Menu+Firm key #2**.

Each line of the diagnostics list can hold up to 80 characters, and the list can hold up to 200 lines. When a new message is logged, it is added on top of the list. When more than 200 lines are stored, the oldest ones are discarded. When the same message is logged multiple times within one second, it is logged only once.

Class methods

`log()`	Add a message to the diagnostics log.

A.3.1. Class methods

A.3.1.1. Diagnostics.log()

Synopsis

`Diagnostics.log(s)`

Purpose

Add a message to the diagnostics log.

Parameters

s	`String` The message text to be displayed.

Exceptions

None.

Additional info

The message will be truncated to fit on one line of the diagnostics widget. The new message will be added on top of the list.

Example

Example A.9. Diagnostics.log()

```
Diagnostics.log("extender " + i + " does not respond");
```

A.4. Extender class

The Extender class provides an interface to a RF extender, including its input ports, serial ports and relay outputs. The extender configuration is read from the configuration file, so in order to be able to control an extender from a script, it needs to be properly defined in the editor. This means that it should be marked as selected in the Extender tab of the System Properties of the configuration file.

Instance Properties

`input`	The `input[]` array contains the power sense inputs of an extender. Normally, a serial extender has 4 power sense inputs. The inputs are numbered from 0 to 3. A wireless extender has no power sense inputs. This can be checked by comparing the array elements with `null`.
`relay`	Array giving access to a specific extender relay port. A serial extender has 4 relay outputs numbered from 0 to 3. A wireless extender has no relay ports, so the array elements will be `null`.
`serial`	This array gives access to the serial port with the specified number of an extender. A serial extender has 4 serial ports numbered from 0 to 3. A wireless extender has no serial ports and the array elements will be `null`.

A.4.1. Instance properties

A.4.1.1. Extender.input

Purpose

The `input[]` array contains the power sense inputs of an extender. Normally, a serial extender has 4 power sense inputs. The inputs are numbered from 0 to 3. A wireless extender has no power sense inputs. This can be checked by comparing the array elements with `null`.

Read/Write : R

Value : Input

Instance of the specified extender input, or `null` if the extender is not defined as a serial extender.

Example

Example A.10. Extender.input

```
// Get input port 0 on extender 0:
var p = CF.extender[0].input[0];
```

A.4.1.2. Extender.relay

Purpose

Array giving access to a specific extender relay port. A serial extender has 4 relay outputs numbered from 0 to 3. A wireless extender has no relay ports, so the array elements will be `null`.

Read/Write : R

Value : Relay

Instance of the specified extender relay port, or `null` if the extender is not serial or the port number is out of range.

Additional info

Refer to Section A.9, “Relay class” for more details on how to control the extender relays.

Example

Example A.11. Extender.relay

```
// Get relay port 0 on extender 0:
var p = CF.extender[0].relay[0];
```

A.4.1.3. Extender.serial

Purpose

This array gives access to the serial port with the specified number of an extender. A serial extender has 4 serial ports numbered from 0 to 3. A wireless extender has no serial ports and the array elements will be `null`.

Read/Write : R

Value : Serial

Instance of the specified extender serial port, or `null` if the extender is not serial or the port number is out of range.

Additional info

Refer to Section A.10, “Serial class” for more details on the extender serial ports.

Example

Example A.12. Extender.serial

```
// Get access to serial port 0 of extender 0:
var p = CF.extender[0].serial[0];
```

A.5. GUI class

Control the graphical user interface of the control panel and access the objects that are displayed on the screen.

Class methods

`alert()`	Display a modal dialog box.
`getDisplayDate()`	Get the control panel date.
`getDisplayTime()`	Get the control panel time.
`updateScreen()`	Force a screen update.
`widget()`	Search for a widget that is currently displayed on the screen. This also includes firm keys and hard keys.

A.5.1. Class methods

A.5.1.1. GUI.alert()

Synopsis

`GUI.alert(`*`message`*`)`

Purpose

Display a modal dialog box.

Parameters

`message`	`String` Message to be shown in the popup box.

Exceptions

None.

Additional info

The dialog box shown is not customizable; it is always shown with a single OK button. For dialog boxes which integrate better with a customized look and feel of a project, it is suggested to use regular panels and buttons. The `alert` method is intended more as a facility to be used during development of a script (for example, to catch unexpected exceptions).

Note

The `GUI.alert()` method is not a blocking call; even though the dialog box shown is modal (i.e., all other user interaction is suspended until the OK button is pressed), the script will resume execution immediately after calling this method.

Example

Example A.13. GUI.alert()

Catching an exception:

```
try {
   GUI.widget("P1").color = 0x0000ff;
} catch (e) {
   GUI.alert(e.name + ":\n" + e.message);
}
```

A.5.1.2. GUI.getDisplayDate()

Synopsis

`GUI.getDisplayDate()`

Purpose

Get the control panel date.

Parameters

None.

Return

`String`

Contains the date as shown in the Date status widget.

Exceptions

None.

Additional info

This method returns a string representation of the date, taking into account the time zone and daylight savings settings as set in ProntoEdit Professional. This is distinct from the date obtained with the Core JavaScript `Date` class, which always operates in UTC on the control panel.

A.5.1.3. GUI.getDisplayTime()

Synopsis

`GUI.getDisplayTime()`

Purpose

Get the control panel time.

Parameters

None.

Return

`String`

Contains the time as shown in the Time status widget.

Exceptions

None.

Additional info

This method returns a string representation of the time, taking into account the time zone and daylight savings settings as set in ProntoEdit Professional. This is distinct from the time obtained with the Core JavaScript `Date` class, which always operates in UTC on the control panel.

A.5.1.4. GUI.updateScreen()

Synopsis

```
GUI.updateScreen()
```

Purpose

Force a screen update.

Parameters

None.

Exceptions

None.

Additional info

Because during script execution the screen is not updated, an explicit screen update can be enforced with this function call. Script execution is temporarily stopped until the screen update is finished.

A.5.1.5. GUI.widget()

Synopsis

```
GUI.widget(tagW)
```

Purpose

Search for a widget that is currently displayed on the screen. This also includes firm keys and hard keys.

Parameters

tagW	`String`

The tag of the widget to search for as defined in the editor.

Return

`Widget`

The class instance corresponding to the found widget, or `null` if not found.

Exceptions

- No argument specified
- Argument is not a string

Additional info

Refer to Section A.13, “Widget class” for detailed information on the return type.

A.6. Image class

This class represents an image in the configuration file or on the screen. It is used when creating dynamically an image or retrieving the image from a button, panel or firm key in order to copy it to another button, panel or firm key.

This can be useful when creating gallery pages with artwork widgets or when creating animated widgets with a changing image.

Instance Properties

`height`	Get the vertical size of the image in pixels.
`width`	Get the horizontal size of the image in pixels.

A.6.1. Image class constructor

A.6.1.1. Image()

Purpose

Create a new Image instance.

Parameters

s	`String` The image creator supports PNG, JPG and uncompressed standard BMP format. No exceptions are thrown if data can not be interpreted correctly.

Return

`Image`

A new Image class instance.

Exceptions

None.

Additional info

The parameter must be raw bitmap data stored as a `String`. This means that it is not the filename of the image.

Example

Example A.14. Image. constructor

```
var myImage = new Image(bitmapdata);
```

A.6.2. Instance properties

A.6.2.1. Image.height

Purpose

Get the vertical size of the image in pixels.

Read/Write : R

Value : Integer

Applicable for

Button, Firm key, Panel

Additional info

The DPI of the image is not used.

A.6.2.2. Image.width

Purpose

Get the horizontal size of the image in pixels.

Read/Write : R

Value : Integer

Applicable for

Button, Firm key, Panel

Additional info

The DPI of the image is not used.

A.7. Input class

This class represents a power sense input port on a serial extender.

Instance Properties

`onData`	Define the callback function for extender input port data.
`onError`	Define the callback function for extender input port errors.
`onTimeout`	Define the callback function when a timeout occurs during an asynchronous `match()` or `wait()` operation.

Instance methods

`get()`	Get the value of the power sense input.
`match()`	Wait for the port state to match a specific state. The operation completes as soon as the port is in the requested state or when the indicated time has passed.
`wait()`	Wait for an input port to change state. The operation completes as soon as the port state changes or when the indicated time has passed.

A.7.1. Instance properties

A.7.1.1. Input.onData

Purpose

Define the callback function for extender input port data.

Read/Write : RW

When assigned, the callback will remain defined as long as the current activity remains active.

Value : OnInputDataCallback

Set to `null` for synchronous (blocking) operation.

A.7.1.2. Input.onError

Purpose

Define the callback function for extender input port errors.

Read/Write : RW

Persistent as long as the current activity remains active.

Value : onInputErrorCallback

Set to a valid function or to `null` if no error handling is desired.

Additional info

In case of an erroneous `match()` or `write()` operation, the `onError` function is called.

A.7.1.3. Input.onTimeout

Purpose

Define the callback function when a timeout occurs during an asynchronous `match()` or `wait()` operation.

Read/Write : RW

Persistent as long as the current activity remains active.

Value : onInputTimeoutCallback

A.7.2. Callback functions

The prototypes of the call back functions are listed below. In the call back functions, you can use '`this`' to refer to the scope of the actual input object that is causing the call back.

A.7.2.1. *`onInputDataCallback`*

Purpose

Called when an asynchronous `match()` or `wait()` completes.

Parameters

`state` `Boolean`

The state of the power sense input: `true` if high, `false` if low.

A.7.2.2. *`onInputErrorCallback`*

Purpose

Called when an error occurs during an asynchronous `get()`, `match()` or `wait()` operation.

Parameters

`e` `PanelError`

The error that occurred as an `Error` object

Example

Example A.15. *`onInputErrorCallback`*

```
// The error string can be retrieved by casting e to a string:
System.print( e );
```

A.7.2.3. *`onInputTimeoutCallback`*

Purpose

Called when a timeout occurs during an asynchronous `match()` or `wait()` operation.

Parameters

None.

A.7.3. Instance methods

Note

One extender can only reference one request at the same time. The below methods will fail and throw an exception when the extender is busy with another request. Therefore avoid using long timeout values!

A.7.3.1. Input.get()

Synopsis

`input.get()`

Purpose

Get the value of the power sense input.

Parameters

None.

Return

`Boolean`

`true` if the input is high, `false` if the input is low.

Exceptions

- Failed (extender error)

Additional info

The `get()` is executed as a blocking call, i.e. script execution continues only after the extender has replied with the requested power sense value.

A.7.3.2. Input.match()

Synopsis

`input.match(state,timeout)`

Purpose

Wait for the port state to match a specific state. The operation completes as soon as the port is in the requested state or when the indicated time has passed.

Parameters

state	`Boolean` The requested state to wait for.
timeout	`Integer` The maximum time in milliseconds to wait for the specified state.

Return

`Boolean`

`true` if port state changed in time, `false` otherwise.

Exceptions

- Not enough arguments specified
- Argument is not a positive integer number
- Failed (extender error)

Additional info

If no `onData` function is specified, the script execution is halted until the operation completes.

Otherwise, the script continues execution and the `onData` function is called when the operation completes. In case of a timeout, the `onTimeout` callback function is invoked instead. Exceptions are passed to the `onError` callback.

A.7.3.3. Input.wait()

Synopsis

`input``.wait(`*`timeout`*`)`

Purpose

Wait for an input port to change state. The operation completes as soon as the port state changes or when the indicated time has passed.

Parameters

`timeout`	`Integer` The maximal time in milliseconds to wait for the specified port to change state.

Return

`Boolean`

`true` if the port state was changed, or `false` if timeout.

Exceptions

- No argument specified
- Argument is not a positive integer number
- Failed (extender error)

Additional info

If no `onData` callback function is specified, script execution is halted until the operation completes. Otherwise, the script continues execution and the specified `onData` function is called when the operation completes. If a timeout occurs, the `onTimeout` function is called instead. The `onError` function is called in case of an exception.

A.8. Page class

This class allows access to the properties of a page in the configuration file.

Instance Properties

`label`	The name of the page as defined in the editor.
`repeatInterval`	This member stores the time after which the page script is repeated.
`tag`	Get the tag of the page.

Instance methods

`widget()`	Searches the page for a specific button or panel and returns the corresponding Widget class instance.

A.8.1. Instance properties

A.8.1.1. Page.label

Purpose

The name of the page as defined in the editor.

Read/Write : R

Value : String

Additional info

The page name is not visible on the control panel, but it can be defined in the editor.

A.8.1.2. Page.repeatInterval

Purpose

This member stores the time after which the page script is repeated.

Read/Write : RW

The page repeat interval can only be set for the current page.

Value : Integer

Page script repeat interval in milliseconds. If the value is zero, the page script is not repeatedly executed.

A.8.1.3. Page.tag

Purpose

Get the tag of the page.

Read/Write : R

Value : String

String containing the page tag.

Additional info

The tag is used to find the page in the configuration file.

A.8.2. Instance methods

A.8.2.1. Page.widget()

Synopsis

`page``.widget(`*`tagW`*`)`

Purpose

Searches the page for a specific button or panel and returns the corresponding Widget class instance.

Parameters

`tagW`	`String` Tag name of the widget to search for.

Return

`Widget`

Class instance corresponding to the first matching widget in the page, or `null` if the widget is not found.

Exceptions

- Not enough arguments specified
- Argument is not a string

Additional info

Refer to Section A.13, “Widget class” for detailed information on the page class members.

Example

Example A.16. Page.widget()

`p.widget("RESULT")`: searches the widget tagged "`RESULT`" on page `p`.

A.9. Relay class

A relay port of a serial extender can be controlled with this class type.

Instance methods

`get()`	Inspect the actual value of a relay output.

`set()`	Set a relay output in a specific state.
`toggle()`	Change the relay output state. If the relay was closed, it is opened. If it was open, it is closed.

A.9.1. Instance methods

A.9.1.1. Relay.get()

Synopsis

`relay``.get()`

Purpose

Inspect the actual value of a relay output.

Parameters

None.

Return

`Boolean`

`true` if the relay is closed, `false` otherwise.

Exceptions

- Failed (extender error)

Additional info

The `get()` is executed as a blocking call, i.e. script execution continues only after the extender has replied with the requested relay state.

A.9.1.2. Relay.set()

Synopsis

`relay``.set(`*`state`*`)`

Purpose

Set a relay output in a specific state.

Parameters

`state`	`Boolean` Set to `true` if the relay should be closed, `false` if it should be open.

Exceptions

- Failed (extender error)

Additional info

The `set()` is executed as a blocking call, i.e. script execution continues only after the extender has performed the requested operation.

A.9.1.3. Relay.toggle()

Synopsis

```
relay.toggle()
```

Purpose

Change the relay output state. If the relay was closed, it is opened. If it was open, it is closed.

Parameters

None.

Exceptions

- Failed (extender error)

Additional info

The `toggle()` is executed as a blocking call, i.e. script execution continues only after the extender has performed the requested operation.

A.10. Serial class

A serial port of an extender can be used to send or receive data. A serial port has its own input buffer on the extender. This buffer accumulates incoming data until the control panel issues a `receive()` command. When receiving data on the serial port, the received bytes will be removed from the input buffer, so that they will not be read twice. When sending data on the serial port, its input buffer will be flushed. Please take in mind that an empty string parameter does not clear the input buffer, while a non empty string does.

Send and receive operations can be combined into one combined `receive()` command in order to support multiple control panels querying for data.

Instance Properties

`bitrate`	Set the serial communication speed.
`databits`	Set the number of data bits for the serial communication.
`onData`	Define the function that is called when data is received after a successful call to `receive()` or `match()`.
`onError`	Define the function that is called when an error occurs during `receive()` or `match()`.
`onTimeout`	Define the callback function when a timeout occurs during an asynchronous `receive()` or `match()`.
`parity`	Set the parity of the serial communication.
`stopbits`	Define the number of stop bits for the serial communication.

Instance methods

`match()`	First transmit an optional string on the serial port to query for data and then start receiving on the same port.
`receive()`	First transmit an optional string on the serial port to query for data and then start receiving on the same port.
`send()`	To transmit data on the serial port using the communication settings as specified in the above data members.

A.10.1. Instance properties

A.10.1.1. Serial.bitrate

Purpose

Set the serial communication speed.

Read/Write : RW

Value : Integer

Valid values are: 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 57600 and 115200 bits per second.

A.10.1.2. Serial.databits

Purpose

Set the number of data bits for the serial communication.

Read/Write : RW

Value : Integer

Valid values are 7 and 8.

A.10.1.3. Serial.onData

Purpose

Define the function that is called when data is received after a successful call to `receive()` or `match()`.

Read/Write : RW

Value : onSerialDataCallback

Set to `null` for synchronous (blocking) operation.

Additional info

If an `onData` function is defined but `onTimeout` is `null`, then in case of a timeout the `onData` callback will be called with the received data.

A.10.1.4. Serial.onError

Purpose

Define the function that is called when an error occurs during `receive()` or `match()`.

Read/Write : RW

Value : onSerialErrorCallback

Set to `null` if no error handling is desired.

A.10.1.5. Serial.onTimeout

Purpose

Define the callback function when a timeout occurs during an asynchronous `receive()` or `match()`.

Read/Write : RW

Persistent as long as the current activity remains active.

Value : onSerialTimeoutCallback

Additional info

If omitted, the `onData` callback will be called with the received data.

A.10.1.6. Serial.parity

Purpose

Set the parity of the serial communication.

Read/Write : RW

Value : Integer

Valid values are: 0 (none), 1 (odd) and 2 (even).

A.10.1.7. Serial.stopbits

Purpose

Define the number of stop bits for the serial communication.

Read/Write : RW

Value : Integer

Valid values are 1 and 2.

A.10.2. Callback functions

The prototypes of the call back functions are as follows. In the call back functions you can use '`this`' to refer to the scope of the Serial object that is causing the call back.

A.10.2.1. *`onSerialDataCallback`*

Purpose

Called when an asynchronous `receive()` or `match()` completes successfully.

Parameters

s	`String` The data that was received on the serial port.

Additional info

This string can contain binary data.

Use s.length to get the number of bytes received.

A.10.2.2. *`onSerialErrorCallback`*

Purpose

Called when an error occurs during an asynchronous `receive()` or `match()`.

Parameters

e	`PanelError` An instance of the PanelError class for the error that occurred.

Example

Example A.17. *`onSerialErrorCallback`*

The error string can be retrieved by casting e to a string:

```
System.print( e );
```

A.10.2.3. *`onSerialTimeoutCallback`*

Purpose

Called when a timeout occurs.

Parameters

s	`String` The partial data that was received on the serial port.

Additional info

This string can contain binary data. Use s.length to get the number of bytes received.

A.10.3. Instance methods

Note

One extender can only reference one request at the same time. The methods below will fail and throw an exception when the extender is busy with another request. Therefore, avoid using long timeout values!

A.10.3.1. Serial.match()

Synopsis

```
serial.match(s,terminator,timeout)
```

Purpose

First transmit an optional string on the serial port to query for data and then start receiving on the same port.

Parameters

s	`String` String to be transmitted, may be `null` or empty.
terminator	`String` The terminator string to wait for.
timeout	`Integer` The maximal time in milliseconds to wait for the serial data to arrive.

Return

`String`

The received data including the terminator string, or an empty string in case of asynchronous operation.

Exceptions

- Argument is not a string
- Argument is not a positive integer number
- Failed (extender error)

Additional info

The operation is complete if the specified terminator string is received or until timeout milliseconds have passed. In the last case the currently received data will be returned.

If no `onData` function is specified, the script execution is halted until the operation completes and the received data is returned. Otherwise, the script continues execution and the specified `onData` function is called when the operation completes.

A.10.3.2. Serial.receive()

Synopsis

`serial.receive(s,count,timeout)`

Purpose

First transmit an optional string on the serial port to query for data and then start receiving on the same port.

Parameters

s	`String` String to be transmitted, may be `null` or empty.
count	`Integer` The number of bytes to receive.
timeout	`Integer` The maximal time in milliseconds to wait for the serial data to arrive.

Return

`String`

The received data, or an empty string in case of asynchronous operation. Can contain binary data.

Exceptions

- Argument is not a positive integer number
- Failed (extender error)

Additional info

The operation is complete if count bytes are received or until timeout milliseconds have passed. In the last case less than count bytes will be returned.

If no `onData` function is specified, the script execution is halted until the operation completes and the received data is returned. Otherwise, the script continues execution and the specified `onData` function is called when the operation completes.

A.10.3.3. Serial.send()

Synopsis

`serial.send(s)`

Purpose

To transmit data on the serial port using the communication settings as specified in the above data members.

Parameters

s	`String`

The data to be transmitted. May contain binary data. Maximal length is 512 bytes.

Exceptions

- Not enough arguments specified
- Argument is not a string
- Failed (extender error)

Additional info

The send is executed as a synchronous (blocking) operation. Script execution is halted until the extender replies that the requested operation is completed.

A.11. System class

The system class gives access to some general system level functionality. Furthermore it manages global information that needs to be shared between different activities. This information is stored as a list of name-value string pairs. The string values can contain binary data. The length is restricted by the available amount of memory.

Class methods

`delay()`	Wait for a specific time. This blocks script execution during the specified time.
`getGlobal()`	Retrieve a string value stored in the global variables list.
`getModel()`	Obtain the control panel model name.
`getApplicationVersion()`	Obtain the control panel application version.
`getBootloaderVersion()`	Obtain the control panel boot loader version.
`getFirmwareVersion()`	Obtain the control panel firmware version.
`getIRVersion()`	Obtain the control panel infrared software version.
`getSerial()`	Obtain the control panel serial number.
`getFreeCFMemory()`	Obtain the percentage of free storage space available for a configuration file.
`include()`	Causes a library script to be included. This library script will be executed, causing the classes and variables declared in that library script to become available in the global scope.
`print()`	Display a debug message on the debug output panel.
`setGlobal()`	Store a string item in the global variables list.

A.11.1. Class methods

A.11.1.1. System.delay()

Synopsis

`System.delay(`*`duration`*`)`

Purpose

Wait for a specific time. This blocks script execution during the specified time.

Parameters

duration	`Integer` Duration of the delay in milliseconds.

Exceptions

- Not enough arguments specified
- Argument is not an integer

Additional info

The screen contents will not be refreshed during a delay. If this is desired, use the `scheduleAfter` function instead.

A.11.1.2. System.getGlobal()

Synopsis

`System.getGlobal(`*`name`*`)`

Purpose

Retrieve a string value stored in the global variables list.

Parameters

name	`String` The name of the global variable to find.

Return

`String`

The value of the global variable, or `null` if the name is not found.

Exceptions

- Not enough arguments specified
- Invalid name

A.11.1.3. System.getModel()

Synopsis

`System.getModel()`

Purpose

Obtain the control panel model name.

Parameters

None.

Return

```
String
```

Control panel model name, e.g. "`TSU9600`"

Exceptions

None.

A.11.1.4. System.getApplicationVersion()

Synopsis

```
System.getApplicationVersion()
```

Purpose

Obtain the control panel application version.

Parameters

None.

Return

```
String
```

Control panel firmware version, e.g. "`7.1.2`"

Exceptions

None.

A.11.1.5. System.getBootloaderVersion()

Synopsis

```
System.getBootloaderVersion()
```

Purpose

Obtain the control panel boot loader version.

Parameters

None.

Return

```
String
```

Control panel boot loader version, e.g. "`BFU1.9.3`"

Exceptions

None.

A.11.1.6. System.getFirmwareVersion()

Synopsis

```
System.getFirmwareVersion()
```

Purpose

Obtain the control panel firmware version.

Parameters

None.

Return

`String`

Control panel firmware version, e.g. "`TSU9600 V2.1`"

Exceptions

None.

A.11.1.7. System.getIRVersion()

Synopsis

```
System.getIRVersion()
```

Purpose

Obtain the control panel infrared software version.

Parameters

None.

Return

`String`

Control panel infrared software version, e.g. "`4.0.20`"

Exceptions

None.

A.11.1.8. System.getSerial()

Synopsis

```
System.getSerial()
```

Purpose

Obtain the control panel serial number.

Parameters

None.

Return

`String`

Control panel serial number, e.g. "`0000063021`"

Exceptions

None.

A.11.1.9. System.getFreeCFMemory()

Synopsis

```
System.getFreeCFMemory()
```

Purpose

Obtain the percentage of free storage space available for a configuration file.

Parameters

None.

Return

`Integer`

A number between 0 (no more storage space available) and 100 (no storage space in use).

Exceptions

None.

A.11.1.10. System.include()

Synopsis

```
System.include(name)
```

Purpose

Causes a library script to be included. This library script will be executed, causing the classes and variables declared in that library script to become available in the global scope.

Parameters

`name`	`String` Filename of the library script.

Exceptions

None.

A.11.1.11. System.print()

Synopsis

`System.print(s)`

Purpose

Display a debug message on the debug output panel.

Parameters

s	`String` Text to be displayed. This text is appended to the label of the debug window. Maximum length: 99 characters. If longer, will be truncated.

Exceptions

None.

Additional info

The debug panel is a panel or button tagged "_PS_DEBUG_". When defining this panel in the editor, make sure it has the text alignment set to bottom left, so that the newly added text always is visible.

Use "`\n`" to insert line breaks in the text output.

A.11.1.12. System.setGlobal()

Synopsis

`System.setGlobal(name)`

`System.setGlobal(name,value)`

Purpose

Store a string item in the global variables list.

Parameters

name	`String` The name under which to store the string value.
value	`String` The string value to store. May contain binary data. The current value associated with the given name, if any, is overwritten. If the new value is `null`, empty or omitted, the current string item with the specified name is removed.

Exceptions

- No argument specified
- Argument is an invalid name
- Insufficient internal memory available

A.12. TCPSocket class

A network socket can be created to establish a TCP connection over a wireless network.

Instance Properties

`connected`	Check the connection state of the socket.
`onClose`	Define the asynchronous socket close callback function.
`onConnect`	Define the asynchronous socket connect callback function.
`onData`	Define the function to be called when data is available on an asynchronous socket.
`onIOError`	Define the error referencer.

Instance methods

`connect()`	Create a connection on an ip address.
`close()`	Terminate the connection.
`write()`	Write data to a socket.
`read()`	Read data from a socket.

A.12.1. TCPSocket class constructor

A.12.1.1. TCPSocket()

Purpose

Create a new TCPSocket instance.

Parameters

`blocking`	`Boolean` Indicates if the new socket should be blocking (`true`) or not (`false`).

Return

A new TCPSocket class instance.

Exceptions

None.

Additional info

When `true`, creates a synchronous (blocking) socket, i.e. the `connect()` and `read()` functions work synchronous, they will block until the operation is finished. If blocking is `false` (or omitted), the asynchronous implementation with callback functions will be used.

A.12.2. Instance properties

A.12.2.1. TCPSocket.connected

Purpose

Check the connection state of the socket.

Read/Write : R

Value : Boolean

`true` if connected, `false` if not.

Additional info

Set to `true` as soon as the connection is established.

A.12.2.2. TCPSocket.onClose

Purpose

Define the asynchronous socket close callback function.

Read/Write : RW

Value : onTCPSocketCloseCallback

Set to `null` if no notification is required.

Additional info

Used to detect the end of a network transfer or that the socket is closed by the destination.

A.12.2.3. TCPSocket.onConnect

Purpose

Define the asynchronous socket connect callback function.

Read/Write : RW

Value : onTCPSocketConnectCallback

The function to be called.

Additional info

This function is called as soon as the connection is established and the socket was created as asynchronous.

A.12.2.4. TCPSocket.onData

Purpose

Define the function to be called when data is available on an asynchronous socket.

Read/Write : RW

Value : onTCPSocketDataCallback

The function to be called.

Additional info

When the `onData` value is triggered, use the `read()` function to get the data.

A.12.2.5. TCPSocket.onIOError

Purpose

Define the error referencer.

Read/Write : RW

Value : onTCPSocketErrorCallback

The function to be called.

Additional info

This callback function is called when the network layer reports an error. The error number is passed as an integer parameter.

A.12.3. Callback functions

The callback functions will be called in the scope of the socket object instance. For example, in the `onConnect` callback function, a `write()` can be done immediately without having to look up the connected socket instance.

The prototypes of the call back functions are as follows:

A.12.3.1. *`onTCPSocketCloseCallback`*

Purpose

Called when the socket is closed successfully.

Parameters

None.

A.12.3.2. *`onTCPSocketConnectCallback`*

Purpose

Called when a `connect()` operation completes successfully on an asynchronous socket.

Parameters

None.

Additional info

When the `connect()` is successful, the `read()` and `write()` operations can be used on the socket.

A.12.3.3. *`onTCPSocketDataCallback`*

Purpose

Called when data is received on an asynchronous socket.

Parameters

None.

Additional info

The callback function can retrieve the received data using the `read()` function.

A.12.3.4. *`onTCPSocketErrorCallback`*

Purpose

Called when an error occurs on an asynchronous socket.

Parameters

e	`PanelError` An instance of the PanelError class for the error.

Example

Example A.18. ***`onTCPSocketErrorCallback`***

The error string can be retrieved by casting e to a string, e.g. `System.print( e );`

A.12.4. Instance methods

A.12.4.1. TCPSocket.connect()

Synopsis

`tcpsocket``.connect(`*`ip`*`,`*`port`*`,`*`timeout`*`)`

Purpose

Create a connection on an ip address.

Parameters

ip	`String` IP address or host name to connect to.

port	`Integer` Port number to connect to.
timeout	`Integer` Maximum time in milliseconds to establish the asynchronous connection.

Exceptions

- Not enough arguments specified
- Argument is not a string
- Argument is not an integer
- Failed (could not connect)

Additional info

For a synchronous socket, the function returns when the connection is established, or when the connection fails.

For an asynchronous socket, it returns immediately and the `onConnect` function is called as soon as the connection is effective. A connection failure will be reported by a call to the `onIOError` function.

A.12.4.2. TCPSocket.close()

Synopsis

```
tcpsocket.close()
```

Purpose

Terminate the connection.

Parameters

None.

Exceptions

- Socket error

A.12.4.3. TCPSocket.write()

Synopsis

```
tcpsocket.write(s)
```

Purpose

Write data to a socket.

Parameters

s	`String`

The data to be transmitted. May contain binary data.

Exceptions

- Not enough arguments specified
- Socket error

Additional info

The string data is queued for output on the network socket.

A.12.4.4. TCPSocket.read()

Synopsis

`tcpsocket``.read(`*`count`*`)`

`tcpsocket``.read(`*`count`*`,`*`timeout`*`)`

Purpose

Read data from a socket.

Parameters

`count` `Integer`

Number of bytes to read.

`timeout` `Integer`

Maximum time in milliseconds to wait for the data to arrive for a synchronous socket. If omitted, returns immediately with the currently available data.

Return

`String`

The available socket data in case of a synchronous socket. For asynchronous sockets, this function returns immediately and the `onData` callback is called when the data is received.

Exceptions

- Argument is not a positive integer number
- Maximum blocking read length exceeded
- Insufficient internal memory available
- Socket error

Additional info

The function reads the available data from the socket. It returns immediately with the read data as result.

This function is typically used in the `onData` callback function to get the received data.

A.13. Widget class

This represents a button or panel in the configuration file or on the screen. This also includes firm keys and hard keys. If the widget is on the current page, the data members will reflect the actual widget properties and they can be adjusted. Otherwise the data members are read-only and reflect the properties as stored in configuration file. The change will be persistent for as long as the activity is active. When changing to another activity and back the widget properties will be reloaded from the configuration file.

Note

During script execution the screen is not updated, so any changes to widget properties will become visible after the script has finished. Refer to Section A.5.1.4, "GUI.updateScreen()" to force intermediate screen updates.

Because the Widget class is used to represent four object types: Button, Firm key, Hard key and Panel, not all properties are meaningful in all cases. In each property description below it is stated for which object type it is applicable.

Note

During the execution of the activity script the current page is not yet created. If you want to manipulate widget properties before they are displayed, please do so in the page script instead.

Instance Properties

`bgcolor` Sets the background color to use if no background image is set.

`color` Sets the foreground (text) color to use.

`height` Determines the vertical size of the widget.

`label` The text displayed in the widget.

`left` Determines the horizontal position of the widget.

`onHold` Contains the function to be called while a button is kept pressed.

`onHoldInterval` Define the button onHold repeat interval time. The default value is 1000, which means that when an `onHold` function is defined, it is called every second.

`onRelease` Program the function to be executed at the next button release.

`stretchImage` Allows stretching the widget image to fit the widget size.

`tag` Get the tag of the widget.

`top` Determines the vertical position of the widget.

`transparent` Controls the transparency of the background if no background image is set.

`visible` Allows hiding or showing a widget on the screen.

`width` Determines the horizontal size of the widget.

Instance methods

`executeActions()`	Executes the action list attached to the button, if any.
`getImage()`	Retrieve the image attached to the widget.
`setImage()`	Change the image of the widget for a specific state (pressed or released).

A.13.1. Instance properties

A.13.1.1. Widget.bgcolor

Purpose

Sets the background color to use if no background image is set.

Read/Write : RW

Value : Integer

Range: 0..16777215

24-bit value (8 bits blue, 8 bits green and 8 bits red); 0xff0000 is blue, 0x00ff00 green, and 0x0000ff is red.

Applicable for

Panel

A.13.1.2. Widget.color

Purpose

Sets the foreground (text) color to use.

Read/Write : RW

Value : Integer

Range: 0..16777215

24-bit value (8 bits blue, 8 bits green and 8 bits red); 0xff0000 is blue, 0x00ff00 green, and 0x0000ff is red.

Applicable for

Panel

A.13.1.3. Widget.height

Purpose

Determines the vertical size of the widget.

Read/Write : RW

Value : Integer

Range: 1 to 65535

Applicable for

Button, Firm key, Panel

Additional info

When the `stretchImage` property of the widget is not set to true, next rules apply:

- If the size is smaller than the height of the displayed image, the image will be clipped.
- If the size is bigger, the remaining space will be transparent.

A.13.1.4. Widget.label

Purpose

The text displayed in the widget.

Read/Write : RW

Value : String

The string can be of any length but the visible part will depend on the dimensions of the widget. May not contain binary data. Use the newline character sequence '\n' to generate a text spanning multiple lines.

Applicable for

Button, Firm key, Panel

A.13.1.5. Widget.left

Purpose

Determines the horizontal position of the widget.

Read/Write : RW

Value : Integer

Range: -32768 to 32767

Applicable for

Button, Firm key, Panel

Additional info

This member stores the number of pixels between the left of the widget and the left side of the screen. Negative values are allowed to place the widget (partly or completely) outside of the screen.

A.13.1.6. Widget.onHold

Purpose

Contains the function to be called while a button is kept pressed.

Read/Write : RW

Value : Function

A valid function, or `null` if no button hold behavior is desired (anymore).

Applicable for

Button, Firm key, Hard key

Additional info

The callback function will be scheduled repeatedly every onHoldInterval milliseconds after the button is pressed, until the button is released.

Note

When a button is pressed for more than 30 seconds, the control panel will automatically release the button. This is to prevent unwanted behaviour because of an object positioned on top of the control panel.

Example

Example A.19. Widget.onHold

```
// Button script showing a counter from 1 to 10 in the
// button label while the button is pressed:
var counter = 1;
onHold = function() {
  label = count++;
  if( count > 10 ) onHold = null;
};
```

A.13.1.7. Widget.onHoldInterval

Purpose

Define the button onHold repeat interval time. The default value is 1000, which means that when an `onHold` function is defined, it is called every second.

Read/Write : RW

Value : Integer

Range: 0…32767

The interval time in milliseconds. If set to 0, the onHold function will not be called anymore.

Applicable for

Button, Firm key, Hard key, Panel

Example

Example A.20. Widget.onHoldInterval

```
// Button script showing a increasing speed counter:
var counter = 1;
var limit = 10;
onHold = function() {
  label = count++;
  if (count == limit) {
    onHoldInterval /= 2;
    limit *= 2;
  }
};
```

A.13.1.8. Widget.onRelease

Purpose

Program the function to be executed at the next button release.

Read/Write : RW

Value : Function

A valid function, or `null` if no button release behavior is desired.

Applicable for

Button, Firm key, Hard key

Additional info

The function will be called once when the button is released.

Example

Example A.21. Widget.onRelease

Example of a button script that changes the button label when the button is released:

```

label = "Pressed";
onRelease = function() { label = "Released"; };
```

A.13.1.9. Widget.stretchImage

Purpose

Allows stretching the widget image to fit the widget size.

Read/Write : RW

Value : Boolean

`true` (stretch) or `false` (don't stretch)

Applicable for

Button, Firm key, Panel

Additional info

When the `stretchImage` property is set, it is applicable for all images that are set in the widget.

For the button, both the pressed and the released state image are set with the `stretchImage` property.

The image is only stretched when it is drawn. This means that when you copy the image, you always get the image in the original size.

A.13.1.10. Widget.tag

Purpose

Get the tag of the widget.

Read/Write : R

Value : String

String containing the tag.

Applicable for

Button, Firm key, Hard key, Panel

Additional info

The tag is used to find the widget in the list of visible widgets or in the configuration file.

A.13.1.11. Widget.top

Purpose

Determines the vertical position of the widget.

Read/Write : RW

Value : Integer

Range: -32768 to 32767

Applicable for

Button, Firm key, Panel

Additional info

The top member contains the number of pixels between the top of the widget and the top of the screen. Negative values are allowed to place the widget (partly or completely) outside of the screen.

A.13.1.12. Widget.transparent

Purpose

Controls the transparency of the background if no background image is set.

Read/Write : RW

Value : Boolean

`true` (transparent) or `false` (opaque)

Applicable for

Panel

Additional info

Note

This property is only supported for panels; setting this property to `true` for a button, will cause that button to not accept touch screen presses (allowing a button below that button to be pressed).

A.13.1.13. Widget.visible

Purpose

Allows hiding or showing a widget on the screen.

Read/Write : RW

Value : Boolean

`true` (visible) or `false` (not visible)

Applicable for

Button, Firm key, Panel

A.13.1.14. Widget.width

Purpose

Determines the horizontal size of the widget.

Read/Write : RW

Value : Integer

Range: 1 to 65535

Applicable for

Button, Firm key, Panel

Additional info

When the `stretchImage` property of the widget is not set to true, next rules apply:

- If the size is smaller than the width of the displayed image, the image will be clipped.
- If the size is bigger, the remaining space will be transparent.

A.13.2. Instance methods

A.13.2.1. Widget.executeActions()

Synopsis

`widget``.executeActions()`

Purpose

Executes the action list attached to the button, if any.

Parameters

None.

Exceptions

None.

Applicable for

Button, Firm key, Hard key

Additional info

This is a blocking function, so script execution will only continue after the action list has been completely finished. If the action list contains a jump to another activity, the script will be aborted.

Note

`executeActions` will fail if another action list is being played already. When executing an activity or page script this is mostly the case. To work around this problem, use scheduleAfter() to execute the actions a little later when the activity switch or page jump is finished.

A.13.2.2. Widget.getImage()

Synopsis

`widget``.getImage()`

`widget``.getImage(`*`index`*`)`

Purpose

Retrieve the image attached to the widget.

Parameters

`index`	`Integer`

The image index. A panel can have only one image (index 0) and a button or firm key can have 2 images: one for the released state (index 0) and one for the pressed state (index 1). If index is omitted, 0 is assumed.

Return

`Image`

An instance of the Image class representing the specified image of the widget.

Exceptions

- Index is out of range

Applicable for

Button, Firm key, Panel

Additional info

An image cannot be created in a script. Instead you can copy it from one widget to another.

Example

Example A.22. Widget.getImage()

Get an image from a panel in the gallery page of the current activity:

```
var img = CF.widget("IMAGE123", "GALLERY").getImage();
```

A.13.2.3. Widget.setImage()

Synopsis

`*widget*.setImage(*img*)`

`*widget*.setImage(*img*,*index*)`

Purpose

Change the image of the widget for a specific state (pressed or released).

Parameters

img	`Image` The image to be assigned to the widget state.
index	`Integer` The image index. A panel can have one image (index 0) and a button or firm key can have 2 images: one for the released state (index 0) and one for the pressed state (index 1). If index is omitted, 0 is assumed.

Exceptions

- Not enough arguments specified

- Argument is not an image.
- Index is out of range.

Applicable for

Button, Firm key, Panel

Additional info

If the size of the new image is bigger than the value of the height and width properties, the image will be clipped. If the size is smaller, the space outside of the image will be transparent.

Example

Example A.23. Widget.setImage()

Example of button showing an animation from the gallery page of the current activity.

It loads the images from the panels tagged `ANIM1_0`, `ANIM1_1` … `ANIM1_9` successively:

```
var count = 0;
onHoldInterval = 100;
onHold = function()
{
  setImage( CF.widget("ANIM1_" + (count % 10), "GALLERY").getImage());
  count++;
};
```

Appendix B. Core JavaScript Classes Description

The following sections list the available Core JavaScript classes in alphabetical order.

Note

This appendix lists all the Core JavaScript classes, methods and properties available in ProntoScript.

Most of these are specified in the [ECMA262] and [ECMA357] standards, while some are non-standard features provided by the script engine used in Pronto control panels.

B.1. Array class

The Array class implements the JavaScript array functionality.

Instance Properties

`length`	Number of elements in the array.

Instance methods

`concat()`	Append elements the the array.
`every()`	Repeat the supplied function for every element of the array, as long as the supplied function returns `true`.
`filter()`	Repeat the supplied function for every element of the array, adding the elements for which this function returns `true` to a newly created array.
`forEach()`	Repeats the supplied function for every element of the array.
`indexOf()`	Index of the supplied item in the array.
`join()`	Create a String by concatenating the elements of the array, with an optional separator in between.
`lastIndexOf()`	Index of the supplied item in the array, counting backwards from the end of the array.
`map()`	Repeat the supplied function for every element of the array, returning an array with the results of each invocation.
`push()`	Add an element to the end of the array.
`pop()`	Return and remove the last element of the array.
`reverse()`	Reverse the order of the elements in the array.
`shift()`	Return and remove the first element of the array.
`slice()`	Return a new array containing a range of elements of the array.
`some()`	Repeats the supplied function for every element of the array, as long as the supplied function returns `false`.

`sort()`	Sort the array elements.
`splice()`	Add and delete array elements.
`unshift()`	Add an element in the beginning of the array.

B.2. Boolean class

Representation of a boolean value

B.3. Date class

Representation of a date/time instance.

Note

In ProntoScript, the time obtained with the `Date` is not the same as the user-visible time.

The user-visible time, which can be obtained using the `GUI.getDisplayTime()` and `GUI.getDisplayDate()` methods, takes the time zone into account, and can change backwards and forwards during script execution (using the settings mode of the contol panel).

On the Pronto panel, the time obtained with the Core JavaScript `Date` class has no relation with the user-visible time and always operates in UTC (Coordinated Universal Time). This time is also not affected by time changes made by the end-user in settings mode, and can thus be used for timers and timeouts, as it will only ever increase, never decrease.

Class methods

`now()`	Return the number of milliseconds since January 1st, 1970 (UTC), until the current JavaScript time.
`parse()`	Convert a string representation of a date or time into the number of milliseconds since January 1st, 1970 (UTC),
`UTC()`	Return number of milliseconds since January 1st, 1970 (UTC), until the current JavaScript time.

Instance methods

`getDate()`	Return the day of the month (UTC) of the time which the instance represents.
`getDay()`	Return the day of the week (UTC) of the time which the instance represents.
`getFullYear()`	Return the year (UTC) of the time which the instance represents.
`getHours()`	Return the hour of the day (in UTC) of the time which the instance represents.
`getMilliseconds()`	Return the seconds compontent (in UTC) of the time which the instance represents.

`getMinutes()`	Return the minutes compontent (in UTC) of the time which the instance represents.
`getMonth()`	Return the month of the year (UTC) of the time which the instance represents.
`getSeconds()`	Return the seconds compontent (in UTC) of the time which the instance represents.
`getTime()`	Return the number of milliseconds since January 1st, 1970 (UTC), of the time which the instance represents.
`getTimezoneOffset()`	In ProntoScript, this always returns 0.
`getUTCDate()`	Return the day of the month (UTC) of the time which the instance represents.
`getUTCDay()`	Return the day of the week (UTC) of the time which the instance represents.
`getUTCFullYear()`	Return the year (UTC) of the time which the instance represents.
`getUTCHours()`	Return the hour of the day (in UTC) of the time which the instance represents.
`getUTCMilliseconds()`	Return the seconds compontent (in UTC) of the time which the instance represents.
`getUTCMinutes()`	Return the minutes compontent (in UTC) of the time which the instance represents.
`getUTCMonth()`	Return the month of the year (UTC) of the time which the instance represents.
`getUTCSeconds()`	Return the seconds compontent (in UTC) of the time which the instance represents.
`getYear()`	Return the number of years since 1900 (UTC), of the time which the instance represents.
`setDate()`	Set the day of the month of the instance time (in UTC)
`setFullYear()`	Set the year of the instance time (in UTC)
`setHours()`	Set the hours of the day of the instance time (in UTC)
`setMinutes()`	Set the minutes component of the instance time (in UTC)
`setMonth()`	Set the month of the year of the instance time (UTC)
`setTime()`	Set the instance time using a specified number of milliseconds since January 1st, 1970 (in UTC).
`setUTCDate()`	Set the day of the month of the instance time (in UTC)
`setUTCFullYear()`	Set the year of the instance time (in UTC)
`setUTCHours()`	Set the hours of the day of the instance time (in UTC)
`setUTCMilliseconds()`	Set the milliseconds component of the instance time (in UTC)

`setUTCMinutes()`	Set the minutes component of the instance time (in UTC)
`setUTCMonth()`	Set the month of the year of the instance time (in UTC)
`setUTCSeconds()`	Set the seconds component of the instance time (in UTC)
`setMilliseconds()`	Set the milliseconds component of the instance time (in UTC)
`setSeconds()`	Set the seconds component of the instance time (in UTC)
`setYear()`	Set the year of the instance time, using the number of years since 1900 (in UTC).
`toDateString()`	Return a string representation of the instance date, in UTC.
`toLocaleDateString()`	Return a string representation of the instance date, in UTC.
`toLocaleFormat()`	Return a string representation of the instance date or time (in UTC), using a specified format.
`toLocaleString()`	Return a string representation of the instance date and time, in UTC.
`toLocaleTimeString()`	Return a string representation of the instance time, in UTC.
`toTimeString()`	Return a string representation of the instance time, in UTC.
`toUTCString()`	Return a string representation of the instance date and time, in UTC.

B.4. Error class

Generic error exception class

Instance Properties

`message`	Message describing the error.
`fileName`	String identifying the script.
`lineNumber`	Line number in the script

B.5. EvalError class

Error exception class for dynamic script evaluation failures.

B.6. Function class

The Function class implements the JavaScript functions, which are special kinds of objects in JavaScript.

Instance Properties

`arguments`	Predefined local array containing the arguments passed to the function.
`arity`	Number of formal parameters.

`caller`	Function object invoking the function.
`length`	Actual number of arguments passed to the function.
`name`	Name of the function.

Instance methods

`apply()`	Invoke a function with an array containing the arguments to be passed to the function.
`call()`	Invoke a function in a specified object context.

B.7. Math class

The `Math` class provides various mathematical functions and constants.

Class Properties

`E`	Constant representing the base of the natural logarithm .
`LOG2E`	Constant representing the base-2 logarithm of the constant E.
`LOG10E`	Constant representing the base-10 logarithm of the constant E.
`LN2`	Constant representing the natural logarithm of 2.
`LN10`	Constant representing the natural logarithm of 10.
`PI`	Mathematical constant π, the ratio of the circumference of a circle to its diameter.
`SQRT2`	The square root of 2
`SQRT1_2`	The square root of 0.5

Class methods

`abs()`	Calculate the absolute value
`acos()`	Arc cosine function
`asin()`	Arc sine function
`atan()`	Arc tangent function
`atan2()`	Arc tangent function (2 variables)
`ceil()`	Ceiling function
`cos()`	Cosine function
`exp()`	Base-E exponential function
`floor()`	Compute the largest integral value not greater than the argument.
`log()`	Natural logarithmic function

`max()`	Return the highest value out of the set of arguments.
`min()`	Return the lowest value out of the set of arguments.
`pow()`	Power function
`random()`	Compute a pseudo-random number
`round()`	Mathematical rounding function
`sin()`	Sine function
`sqrt()`	Square root function
`tan()`	Tangent function

B.8. Namespace class

Represents an XML name space, which can be used to obtain elements from an E4X `XML` object which which are not in the document's default namespace.

Instance Properties

`label`	Namespace prefix
`uri`	Uniform Resource Identifier of the namespace.

B.9. Number class

The `Number` class provides various functions and constants related to numbers.

Class Properties

`NaN`	Representation of the special not-a-number value.
`POSITIVE_INFINITY`	Representation of a positive infinity.
`NEGATIVE_INFINITY`	Representation of a negative infinity.
`MAX_VALUE`	The largest representable number.
`MIN_VALUE`	The smallers representable number.

Instance methods

`toLocaleString()`	Return a string representation of the number instance.
`toFixed()`	Return a string in with a fixed-point representation of the number instance.
`toExponential()`	Return a string with the exponential (scientific) representation of the number instance.
`toPrecision()`	Return a string representation of the number instance, with a specified precision.

B.10. Object class

Base object class, from which all other classes and objects are derived.

Instance Properties

`__count__`	Number of properties of the object
`__parent__`	Object context
`__proto__`	Object prototype at the time of the object instantiation.

Instance methods

`hasOwnProperty()`	Test if a property is defined directly in the object, instead of in the prototype chain.
`isPrototypeOf()`	Test if an object is in the prototype chain of the object instance.
`propertyIsEnumerable()`	Test if a property is an enumerable property of the object.
`toLocaleString()`	Return a string representation of the object
`toSource()`	Return a JavaScript source-code representation of the object
`toString()`	Return a string representation of the object
`unwatch()`	Stop calling the function set with `watch`, whenever a given properties' value changes.
`valueOf()`	Return the primitive value of an object, or the object itself it the object cannot be converted to a primitive value.
`watch()`	Specify a function to be called whenever a given properties' value changes.
`__defineGetter__()`	Specify a function to use when retrieving the value of a given property.
`__defineSetter__()`	Specify a function to use when setting the value of a given property.
`__lookupGetter__()`	Obtain the function set with `__defineGetter__`.
`__lookupSetter__()`	Obtain the function set with `__defineSetter__`.

B.11. QName class

Represents a qualified XML name, as obtained using the `name` method of an E4X XML object instance.

Instance Properties

`localName`	Local name
`uri`	Uniform Resource Identifier

B.12. RangeError class

Error exception class for failures due to parameters which have a value which is outside an allowed range.

B.13. ReferenceError class

Error exception class for failures due to use of a variable which is not defined.

B.14. RegExp class

Representation of a regular expression.

Class Properties

`input` The complete string that was tested in the last regular expression match.

`lastMatch` The text of the last regular expression match.

`lastParen` Last parenthesized substring match of the last regular expression applied.

`leftContext` Substring preceding the last regular expression match.

`rightContext` Substring following the last regular expression match.

`` $` `` Shortcut for the `leftContext` property.

`$'` Shortcut for the `rightContext` property.

`$_` Shortcut for the `input` property.

`$+` Shortcut for the `lastParen` property.

`$&` Shortcut for the `lastMatch` property.

`$1` First parenthesized substring match of the last regular expression applied.

`$2` Second parenthesized substring match of the last regular expression applied.

`$3` Third parenthesized substring match of the last regular expression applied.

`$4` Fourth parenthesized substring match of the last regular expression applied.

`$5` Fifth parenthesized substring match of the last regular expression applied.

`$6` Sixth parenthesized substring match of the last regular expression applied.

`$7` Seventh parenthesized substring match of the last regular expression applied.

`$8` Eight parenthesized substring match of the last regular expression applied.

`$9` Nineth parenthesized substring match of the last regular expression applied.

Instance Properties

`global`	Wether the regular expression should match once (`false`), or for every occurance in the input (`true`).
`ignoreCase`	Wether the regular expression should match case-sensitive (`false`), or case-insensitive (`true`).
`lastIndex`	Index of the last match, also used for determining the start of the next match attempt.
`multiline`	Wether the regular expression should match accross multiple lines (`true`), or not (`false`).
`source`	The regular expression text

Instance methods

`compile()`	Optimize the regular expression for repeated use.
`exec()`	Execute the regular expression on a string, returning a result array.
`test()`	Test if the regular expression matches a string.

B.15. String class

Representation of a text string.

Instance Properties

`length`	Number of characters in the string

Class methods

`fromCharCode()`	Construct a string from one or more Unicode character codes.

Instance methods

`anchor()`	Return the string instance, surrounded with the HTML element tags `<a name=...>` and `</a>`
`big()`	Return the string instance, surrounded with the HTML element tags `<big>` and `</big>`
`blink()`	Return the string instance, surrounded with the HTML element tags `<blink>` and `</blink>`
`bold()`	Return the string instance, surrounded with the HTML element tags `<b>` and `</b>`
`charAt()`	Return the character at the specified index.
`charCodeAt()`	Return the Unicode character code of the character at the specified index.
`concat()`	Append the arguments to the string instance

`fixed()`	Return the string instance, surrounded with the HTML element tags `<tt>` and `</tt>`
`fontcolor()`	Return the string instance, surrounded with the HTML element tags `<font color=...>` and `</font>`
`fontsize()`	Return the string instance, surrounded with the HTML element tags `<font size=...>` and `</font>`
`indexOf()`	Return the character index of the first occurance of a specified string in the string instance
`italics()`	Return the string instance, surrounded with the HTML element tags `<i>` and `</i>`
`lastIndexOf()`	Return the character index of the last occurance of a specified string in the string instance
`link()`	Return the string instance, surrounded with the HTML element tags `<a href=...>` and `</a>`
`localeCompare()`	Compare a string for alphabetic sorting
`match()`	Apply a regular expression to the string instance.
`replace()`	Perform a regular expression search-and-replace.
`search()`	Fast search using a regular expression in the string instance.
`slice()`	Return a substring, specified by start and end indices (of which the latter may be negative to refer to an offset to the end of the string).
`small()`	Return the string instance, surrounded with the HTML element tags `<small>` and `</small>`
`split()`	Return an array of substrings from the string instance, based on a specified separator string.
`strike()`	Return the string instance, surrounded with the HTML element tags `<strike>` and `</strike>`
`sub()`	Return the string instance, surrounded with the HTML element tags `<sub>` and `</sub>`
`substr()`	Return a substring, specified by start index and length.
`substring()`	Return a substring, specified by start and end indices.
`sup()`	Return the string instance, surrounded with the HTML element tags `<sup>` and `</sup>`
`quote()`	Return the string instance, surrounded with quotes
`toLowerCase()`	Return the string instance, replacing all uppercase characters with corresponding lowercase characters.
`toLocaleLowerCase()`	Return the string instance, replacing all uppercase characters with corresponding lowercase characters.
`toLocaleUpperCase()`	Return the string instance, replacing all lowercase characters with corresponding uppercase characters.

`toUpperCase()`	Return the string instance, replacing all lowercase characters with corresponding uppercase characters.

B.16. SyntaxError class

Error exception class for script parsing failures encountered in the compilation of a script fragment.

B.17. TypeError class

Error exception class for failures due to an unexpected type of a value.

B.18. URIError class

Error exception class for failures encountered in URI processing.

B.19. XML class

Representation of an XML document or document fragment.

Class Properties

`ignoreComments`	Wether comments are ignored or not when parsing XML.
`ignoreProcessingInstructions`	Wether processing instructions are ignored or not when parsing XML.
`ignoreWhitespace`	Wether white space is ignored or not when parsing XML.
`prettyPrinting`	Wether XML serialization methods should reformat the XML or not
`prettyIndent`	The amount of spaces to use for indentation when `prettyPrinting` is `true`

Instance methods

`addNamespace()`	Add a namespace to the XML object
`appendChild()`	Append an XML object to the end of the XML object instance.
`attribute()`	Obtain the value of a specified attribute.
`attributes()`	Obtain the attribute values of the XML object instance.
`child()`	Return the children of the XML object instance.
`childIndex()`	Return the index of an XML object within the XML object instance.
`children()`	Obtain the children of the XML object instance, in sequence order.
`comments()`	Obtain the comments of the XML object instance, in sequence order.
`contains()`	Test if an XML object is contained within the XML object instance.

`copy()`	Return a copy of the XML object instance.
`descendants()`	Obtain all XML nodes matching a name, taking into account the node hierarchy.
`elements()`	Obtain the XML elements of an XML object instance.
`hasComplexContent()`	Tests if the XML object instance has multiple layers of XML nodes.
`hasSimpleContent()`	Tests if the XML object instance has only a single layer of XML nodes.
`inScopeNamespaces()`	Return the namespaces relevant to the XML object instance.
`insertChildAfter()`	Add an XML object after a given XML object in the XML object instance.
`insertChildBefore()`	Add an XML object before a given XML object in the XML object instance.
`length()`	Return the number of XML documents in the XML object instance.
`localName()`	Return the local name part of the qualified name of the XML object instance.
`name()`	Return the qualified name of the XML object instance.
`namespace()`	Return the specified namespace of the XML object instance.
`namespaceDeclarations()`	Return the namespace declarations of the XML object instance.
`nodeKind()`	Return the XML node type of the XML node which the XML object instance represents.
`normalize()`	Normalizes the XML object instance.
`parent()`	Return the parent of an XML object instance.
`prependChild()`	Add an XML object to the begin of the XML object instance.
`processingInstructions()`	Obtain the XML processing instructions of an XML object instance.
`removeNamespace()`	Remove a namespace from the XML object instance.
`replace()`	Replace a specified property in the XML object instance.
`setChildren()`	Set the child properties of the XML object instance.
`setLocalName()`	Set the local name part of the XML object instance.
`setName()`	Set the name of the XML object instance.
`setNamespace()`	Set the namespace of the XML object instance.
`text()`	Return the XML text nodes of the XML object instance.
`toXMLString()`	Return a string representation of the XML object instance.

Appendix C. Predefined tags

The tags defined below have a special meaning. Avoid using them for your own widgets.

The following tags are defined for the firm keys:

Firm button	**Tag**
1 (left-most)	`PS_FIRM1`
2	`PS_FIRM2`
3	`PS_FIRM3`
4	`PS_FIRM4`
5 (right-most)	`PS_FIRM5`

Hard button tags:

Hard button	**Tag**
Back	`PS_BACK`
Backlight	`PS_BACKLIGHT`
Channel down	`PS_CHANNEL_DOWN`
Channel up	`PS_CHANNEL_UP`
Cursor down	`PS_CURSOR_DOWN`
Cursor left	`PS_CURSOR_LEFT`
Cursor right	`PS_CURSOR_RIGHT`
Cursor up	`PS_CURSOR_UP`
Guide	`PS_GUIDE`
Home	`PS_HOME`
Info	`PS_INFO`
OK	`PS_OK`
Menu	`PS_MENU`
Mute	`PS_MUTE`
Page down	`PS_PAGE_DOWN`
Page up	`PS_PAGE_UP`
Power	`PS_POWER`
Volume down	`PS_VOLUME_DOWN`
Volume up	`PS_VOLUME_UP`

Predefined activity tags:

Activity	**Tag**
System activity	`PS_SYSTEM`

The system page has also a special tag:

Page	**Tag**
System page	`PS_SYSTEM`

Debug widget tag:

Page	Tag
Debug panel	`_PS_DEBUG_`

Appendix D. Pronto font

The following tables list the contents of the ProntoMaestro font that is available on the control panel. These special unicode characters can be put in a text using the `\u` prefix followed by the four-digit, hexadecimal unicode number.

For example, consider the following button script:

```
label = "Press \uF087; to start the movie";
```

This will put the text "**Press ▶ to start the movie**" on the button label.

Table D.1. Basic Latin font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Table D.2. Supplemental Latin font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00A0		¡	¢	£	€	¥	¦	§	¨	©	ª	«	¬		®	¯
00B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
00D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
00F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
0100	Ā	ā	Ă	ă	Ą	ą	Ć	ć	Ĉ	ĉ	Ċ	ċ	Č	č	Ď	ď
0110	Đ	đ	Ē	ē	Ĕ	ĕ	Ė	ė	Ę	ę	Ě	ě	Ĝ	ĝ	Ğ	ğ
0120	Ġ	ġ	Ģ	ģ	Ĥ	ĥ	Ħ	ħ	Ĩ	ĩ	Ī	ī	Ĭ	ĭ	Į	į
0130	İ	ı	Ĳ	ĳ	Ĵ	ĵ	Ķ	ķ	ĸ	Ĺ	ĺ	Ļ	ļ	Ľ	ľ	Ŀ
0140	ŀ	Ł	ł	Ń	ń	Ņ	ņ	Ň	ň	ŉ	Ŋ	ŋ	Ō	ō	Ŏ	ŏ
0150	Ő	ő	Œ	œ	Ŕ	ŕ	Ŗ	ŗ	Ř	ř	Ś	ś	Ŝ	ŝ	Ş	ş
0160	Š	š	Ţ	ţ	Ť	ť	Ŧ	ŧ	Ũ	ũ	Ū	ū	Ŭ	ŭ	Ů	ů
0170	Ű	ű	Ų	ų	Ŵ	ŵ	Ŷ	ŷ	Ÿ	Ź	ź	Ż	ż	Ž	ž	ſ
0190			ƒ													

Table D.3. Spacing modifier font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
02C0							ˆ	ˇ								

02D0									˘	˙	˚	˛	˜	˝		

Table D.4. Greek font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0390					Δ											
03A0										Ω						
03C0	π															

Table D.5. Cyrillic font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0410	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
0420	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
0430	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
0440	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я

Table D.6. Hebrew font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
05D0	א	ב	ג	ד	ה	ו	ז	ח	ט	י	ך	כ	ל	ם	מ	ן
05E0	נ	ס	ע	ף	פ	ץ	צ	ק	ר	ש	ת					

Table D.7. General Punctuation font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2010				–	—				‘	’	‚		“	”	„	
2020	†	‡	•				…									
2030	‰									‹	›					

Table D.8. Currency font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20A0													€			

Table D.9. Letterlike font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2120			™													

Table D.10. Mathematical operator font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2200			∂													∏
2210		∑	−			∕					√				∞	
2220												∫				
2240									≈							

2260	≠				≤	≥										

Table D.11. Geometrical shape font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
25C0											◊					

Table D.12. Custom font symbols

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
F020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
F030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
F040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
F050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
F060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
F070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	◰
F080	▣	⧈	‚	⤢	„	⏸	■	▶	ˆ	◀	š	⧓	Œ	▦	◳	⚿
F090	⏱	‘	’	“	”	‹	P◂P	›	¨	▷	š	◁	œ	☼	▤	Ÿ
F0A0	⏪	¡	II▶	👤	I-II	⏻	▲▲	⌄	¨	●	⏮	⏏	⌃	▤	⏭	⇕
F0B0	▭	....	-/--	⏩	´	µ	|◀	⌧	¸	☼	⏭	◐	☀	◕	◀II	¿
F0C0	⧐	↔	12	↔	▸PP◂	▭	≡i	▶◀	⏱	⟵	?	◷	📖	⇥	◇	⇨
F0D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	≡▼	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
F0E0	ⓘ	⟳	â	⇕	ä	⇔	▢	ç	è	é	ê	ë	♪	⛶	î	ï
F0F0	🔇	i+	ò	ó	ô	▢+	ö	◀|▶	⇨	ù	ú	û	ü	⌗	ᗡD	dts
FB00		ﬁ	ﬂ													

Further reading

Note

We strongly encourage you to get a copy of the [Flanagan] book! For the Pronto development team it has proven itself as a bible. When giving support to you, it can be most effective to refer to a particular section or example in this book.

[Flanagan] David Flanagan. Copyright © 2006, 2002, 1998, 1997, 1996, O'Reilly & Media, Inc.. Paula Ferguson. 0-596-10199-6. O'Reilly & Media, Inc.. *JavaScript: The Definitive Guide, Fifth Edition*.

[Crockford] Douglas Crockford. Copyright © 2008 O'Reilly & Media, Inc.. 0-596-51774-2. O'Reilly & Media, Inc.. *JavaScript: The Good Parts*. Unearthing the Excellence in JavaScript.

A very useful tool for checking your script for errors:

[JSLint] *JSLint, The JavaScript Verifier* [http://www.jslint.com/] .

A very extensive reference and a guide on the Core JavaScript 1.6, as well as a "re-introduction to JavaScript" can be found at:

[Mozilla] *JavaScript - Mozilla Developer Center* [http://developer.mozilla.org/en/JavaScript] .

ECMA-262 specifies a standardized variant of the JavaScript language, on which ProntoScript builds. This standard documents most of the Core JavaScript features availabe in ProntoScript.

[ECMA262] *Standard ECMA-262, 3rd edition* [http://www.ecma-international.org/publications/standards/Ecma-262.htm] . Ecma International. *ECMAScript Language Specification*.

The E4X support available in ProntoScript is specified by the ECMA-357 standard.

[ECMA357] *Standard ECMA-357* [http://www.ecma-international.org/publications/standards/Ecma-357.htm] . Ecma International. *ECMAScript for XML (E4X) Specification*.

Index