ProntoScript

pronto
<script>

EN Developer's Guide

pronto

PHILIPS

Disclaimer

Conditions to the use of the ProntoScript Developers Guide

1. ProntoScript 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 ProntoScript. 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 ProntoScript (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 hereby 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:

(a) otherwise use the PIP;
(b) modify, adapt, alter, translate, disassemble, re-create, copy, decompile, reverse engineer, or create derivative works from the PSDG, Pronto products and/or the ProntoScript, or;
(c) 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.

Preface

This is the first edition of the ProntoScript Developer's Guide. It is targeted towards programmers who want to *develop* rich 2-way applications for the Pronto Professional Platform. Custom installers who want to *integrate* such 2-way modules into projects are **not** the intended audience. In that case we can offer the "*ProntoScript Installer's Guide*".

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, we refer to David Flanagan's *"Javascript, the Definitive Guide, 5th edition"* published with O'Reilly [Flanagan].

Table of Contents

DISCLAIMER **2**

PREFACE **3**

USING THIS GUIDE 3

TABLE OF CONTENTS **4**

CHAPTER 1. INTRODUCTION **8**

1.1. WHY PRONTOSCRIPT? 8
1.2. A SIMPLE BUTTON SCRIPT 9
1.3. PRONTOSCRIPT FEATURES 10

CHAPTER 2. CORE JAVASCRIPT **12**

2.1. VARIABLES 12
Primitive types *12*
Numbers 12
Strings 12
Boolean 13
Arrays *13*
2.2. OPERATORS 13
Arithmetic operators *14*
Comparative operators *14*
Bitwise operators *15*
2.3. STATEMENT BLOCKS 15
2.4. CONTROL FLOW 15
if/else *15*
switch blocks *16*
while loops *16*
for loops *17*
break statement *17*
2.5. EXCEPTIONS 17
2.6. FUNCTIONS 17
2.7. OBJECTS 18
2.8. REGULAR EXPRESSIONS 18
2.9. MATH OBJECT 18

CHAPTER 3. WIDGETS **19**

3.1. PANELS 20
3.2. BUTTONS 23
3.3. HARD BUTTONS 24
3.4. FIRM KEYS 24

CHAPTER 4. ACTION LISTS **26**

CHAPTER 5. TIMERS **27**

5.1. BLOCKING WAIT 27
5.2. PAGE TIMER 27
5.3. SCHEDULEAFTER() 28
5.3. BEHAVIOR DURING SLEEP MODE 29

CHAPTER 6. LEVELS, SCOPE AND LIFETIME **30**

6.1. LEVELS 30
6.2. SCOPE 30
6.3. LIFETIME 30

CHAPTER 7. ACTIVITIES AND PAGES **31**

7.1. ACTIVITY SCRIPT 31
Usage *31*
Home activity *31*
7.2. PAGE SCRIPT 31
Usage *31*
Page label *32*

Home page ... *32*
Jump to other activity ... *33*
Multiple page jumps within activity ... *33*

CHAPTER 8. EXTENDERS ... **34**

8.1. CF.EXTENDER[] ... 34
8.2. SERIAL PORTS ... 34
8.3. INPUTS ... 36
8.4. RELAYS ... 36
8.5. LIMITATIONS ... 37

CHAPTER 9. NETWORK CONNECTIONS ... **38**

CHAPTER 10. CREATING PRONTOSCRIPT MODULES ... **41**

CHAPTER 11. EXCEPTIONAL SCENARIOS ... **43**

11.1. OUT OF MEMORY ... 43
11.2. NESTED SCRIPTING ... 43
11.3. INFINITE SCRIPTS ... 43
11.4. INVALID ARGUMENTS ... 43
11.5. SCRIPT EXCEPTIONS ... 43

CHAPTER 12. DEBUGGING YOUR SCRIPT ... **45**

12.1. DEBUG WIDGET ... 45
12.2. SYSTEM.PRINT() ... 46

APPENDIX A: PRONTOSCRIPT CLASSES DESCRIPTION (PRONTOSCRIPT API) ... **47**

A.1. ACTIVITY CLASS ... 48
Description ... *48*
Instance properties ... *48*
label ... 48
tag ... 48
Class methods ... *48*
scheduleAfter() ... 48
Instance methods ... *48*
page() ... 48
widget() ... 48
A.2. CF CLASS ... 49
Description ... *49*
Class properties ... *49*
extender[] ... 49
Class methods ... *49*
activity() ... 49
page() ... 49
widget() ... 49
A.3. DIAGNOSTICS CLASS ... 50
Description ... *50*
Class properties ... *50*
Class methods ... *50*
log() ... 50
A.4. EXTENDER CLASS ... 50
Description ... *50*
Instance properties ... *50*
input[] ... 50
relay[] ... 50
serial[] ... 50
Instance methods ... *50*
A.5. GUI CLASS ... 51
Description ... *51*
Class properties ... *51*
Class methods ... *51*
getDisplayDate() ... 51
getDisplayTime() ... 51
updateScreen() ... 51
widget() ... 51
A.6. IMAGE CLASS ... 51
Description ... *51*

Instance properties 51
Instance methods 51
A.7. INPUT CLASS 52
Description 52
Instance properties 52
onData 52
onError 52
onTimeout 52
Callback functions 52
onInputDataCallback 52
onInputErrorCallback 52
onInputTimeoutCallback 52
Instance methods 53
get() 53
match() 53
wait() 53
A.8. PAGE CLASS 54
Description 54
Instance properties 54
label 54
repeatInterval 54
tag 54
Instance methods 54
widget() 54
A.9. RELAY CLASS 55
Description 55
Instance properties 55
Instance methods 55
get() 55
set() 55
toggle() 55
A.10. SERIAL CLASS 56
Description 56
Instance properties 56
bitrate 56
databits 56
onData 56
onError 56
onTimeout 56
parity 56
stopbits 56
Callback functions 57
onSerialDataCallback 57
onSerialErrorCallback 57
onSerialTimeoutCallback 57
Instance methods 57
match() 57
receive() 57
send() 57
A.11. SYSTEM CLASS 58
Description 58
Class properties 58
Class methods 58
delay() 58
getGlobal() 58
getFirmwareVersion() 58
print() 58
setGlobal() 58
A.12. TCPSOCKET CLASS 59
Description 59
Class constructor 59
TCPSocket() 59
Instance properties 59
connected 59
onClose 59
onConnect 59
onData 59
onIOError 59
Callback functions 59

onTCPSocketCloseCallback....59
onTCPSocketConnectCallback....59
onTCPSocketDataCallback....59
onTCPSocketErrorCallback....59
Instance methods....60
connect()....60
close()....60
write()....60
read()....60
A.13. WIDGET CLASS....61
Description....61
Instance properties....61
height....61
label....61
left....61
onHold....61
onHoldInterval....61
onRelease....61
tag....62
top....62
visible....62
width....62
Instance methods....63
executeActions()....63
getImage()....63
setImage()....63

APPENDIX B: PREDEFINED TAGS....64

APPENDIX C: PRONTO FONT....65

FURTHER READING....67

GLOSSARY....68

Chapter 1. Introduction

1.1. Why ProntoScript?

At Philips we took up the challenge to add 2-way communication and dynamic UI's to the Pronto system, to bring it to an ever higher level of home automation sophistication.
We wanted a system that:

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

We concluded that JavaScript, a popular and proven scripting language, is the ideal solution. Integrated into ProntoEdit Professional, it unlocks the full power of the new 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 off 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.

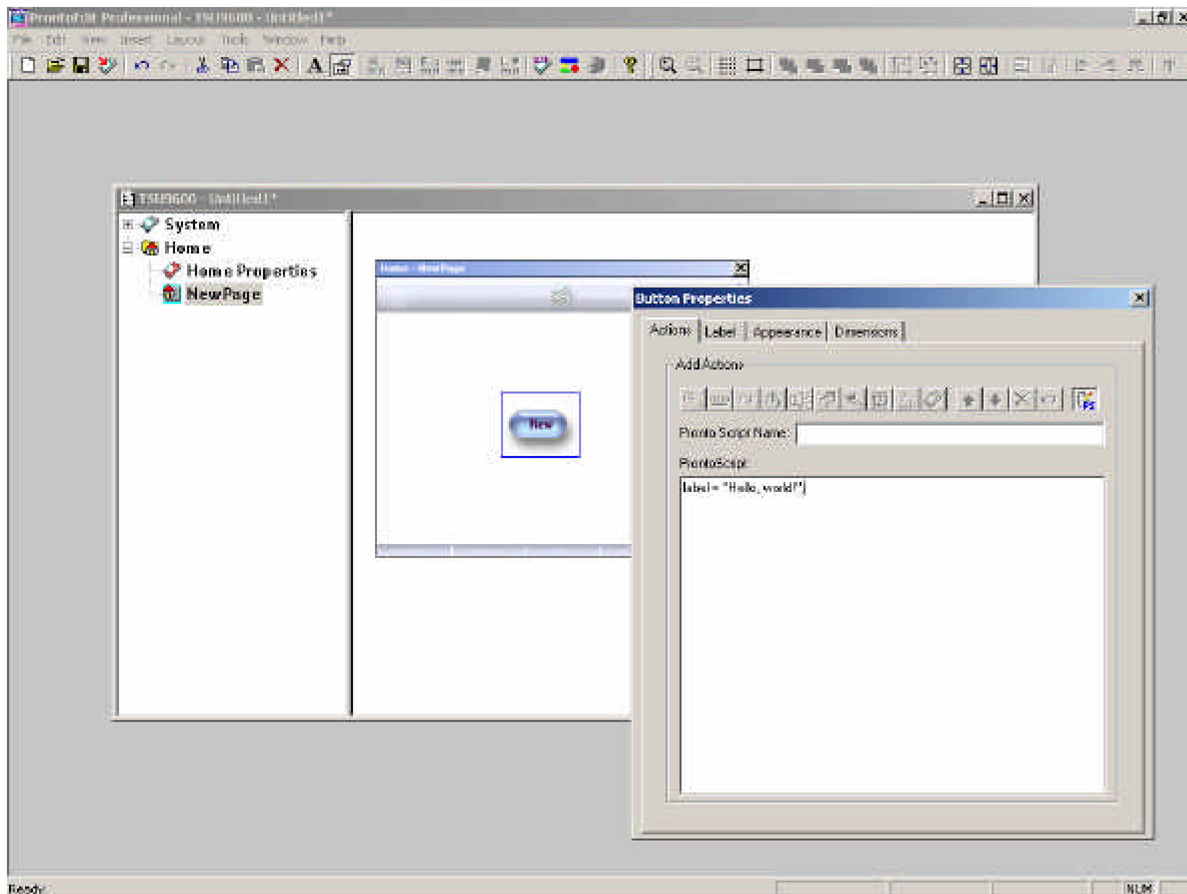
Let's begin our journey with the classic "Hello, world!" program and see how to write this in ProntoScript.

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 we press the button.



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
    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).

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 we would write:

```
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. In a later chapter we will go into great detail with more real live examples for getting the most out of them.

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.

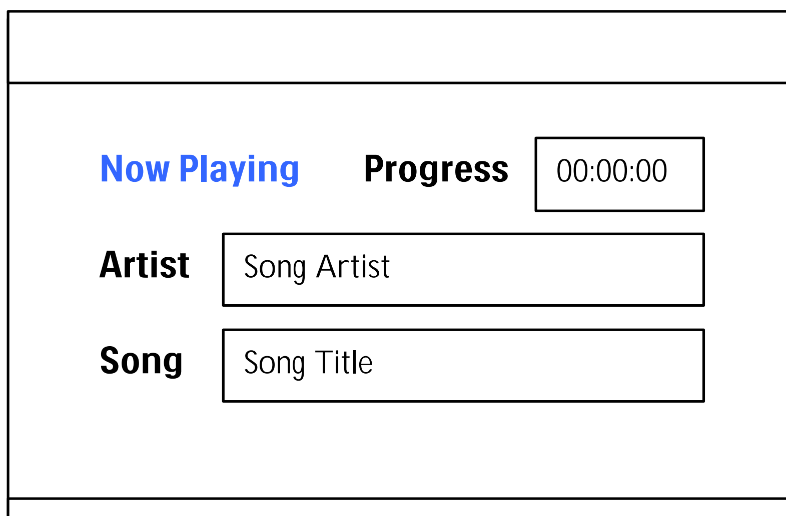
Example: A windows Sideshow gadget transmits these song data in XML format:

```
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:



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

Primitive types

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

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)
```

Strings

A string is a sequence of Unicode characters. The first version of ProntoScript guarantees only ASCII text, more to come.

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
```

Boolean

As in other programming languages, the boolean type is typically used for representing the result of comparisons, e.g. in a 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 & 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
    ...
}
```

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

JavaScripts operators are inspired by the syntax of the C - C++ - Java language syntax family.
For people with experience with these there are little surprises.
We will illustrate with a few examples.

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
```

Comparative operators

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

Assignment Operator =

```
a = 5
```

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

Equality Operator ==
This is the operator that is used to compare for equality. Again (yes, we are repeating ourselves), 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 what you want. If you really want to avoid the type conversion, you should use the Identity operator

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
```

Bitwise operators

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

```
//Bitwise AND
0x1234 & 0x00FF // -> 0x0034: use 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
```

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

if/else

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

The last two lines are optional here.

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

switch blocks

switch (*expression*) {
      statements
}

```
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 that 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 efficient than compile time versions of C, C++ and Java.

while loops

while (*expression*)
      statement

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

for loops

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

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

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

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

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.

```
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 the scripting language JavaScript, functions serve a few purposes

- Define a chunk of functionality but don't execute it yet.
- Execute is at a later stage by calling the function.
- Encapsulate logic into organized, reusable blocks.
- Change the behavior of a particular execution 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: allow 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 = GUI.widget("MY_BUTTON");
var 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.

```
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. Regular Expressions

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

2.9. Math object

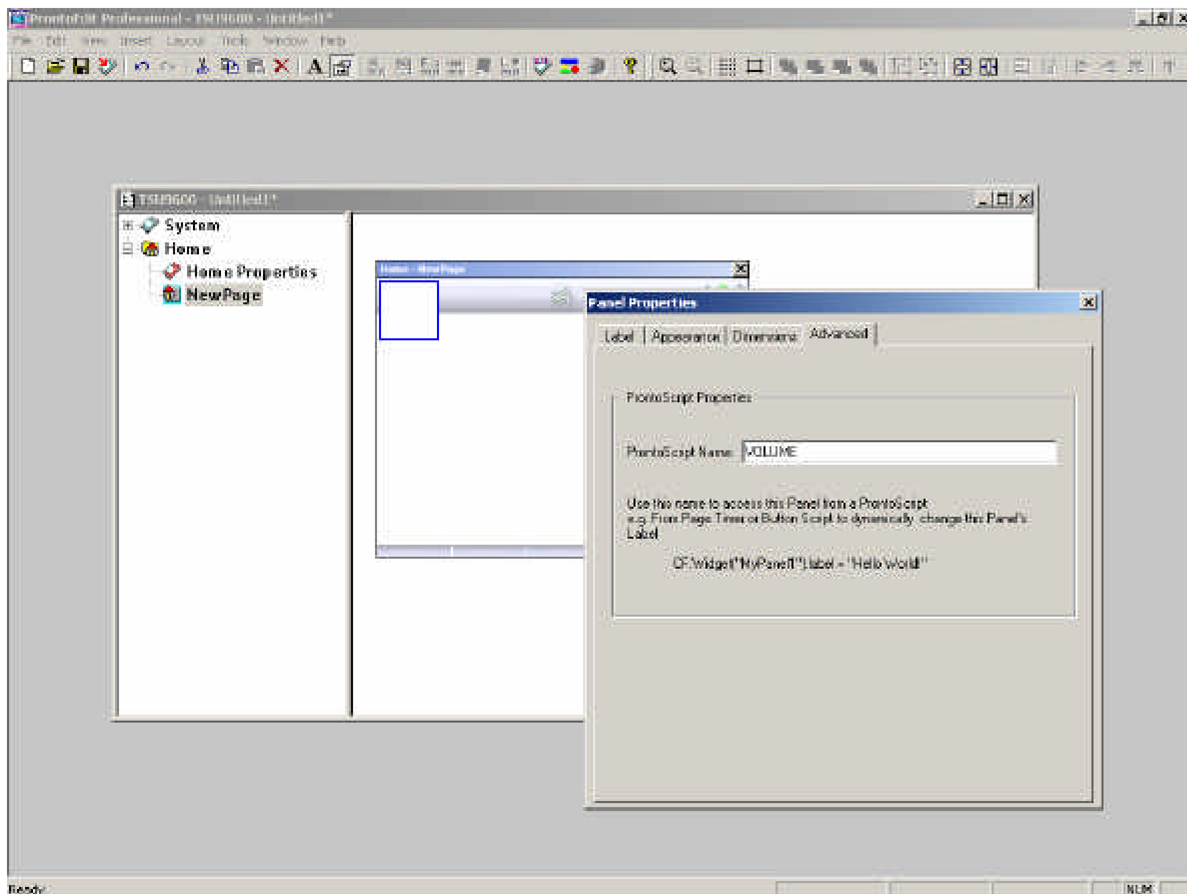
The Math object gives access to a number of useful mathematical constants and functions.

```
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

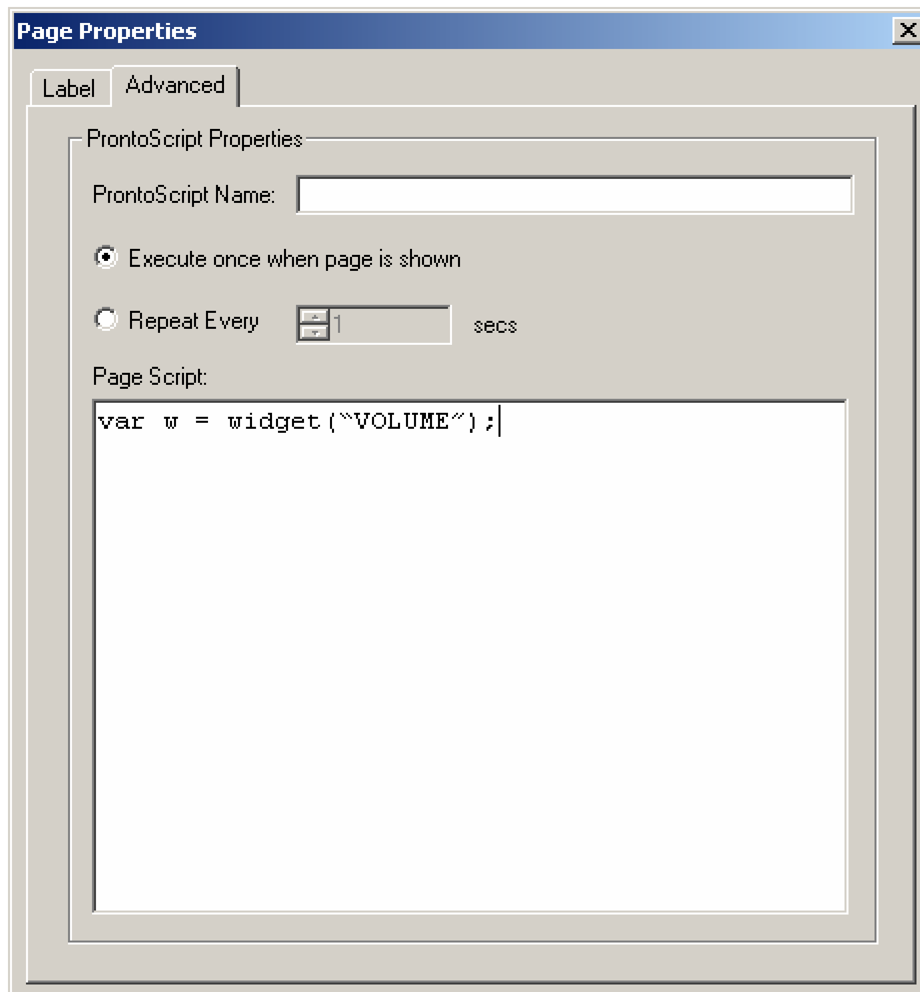
In the editor a page is composed of a number of graphical objects, called widgets. These widgets can be manipulated from a script to create a more dynamic user interface. The most obvious widgets are buttons and panels, but also hard buttons are considered as widgets because they share a number of properties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



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:



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

Note that 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: We propose the convention 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 a text and/or an image. Until now you used it to display a 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". Now you can do the same! Let's guide you through some examples.

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.

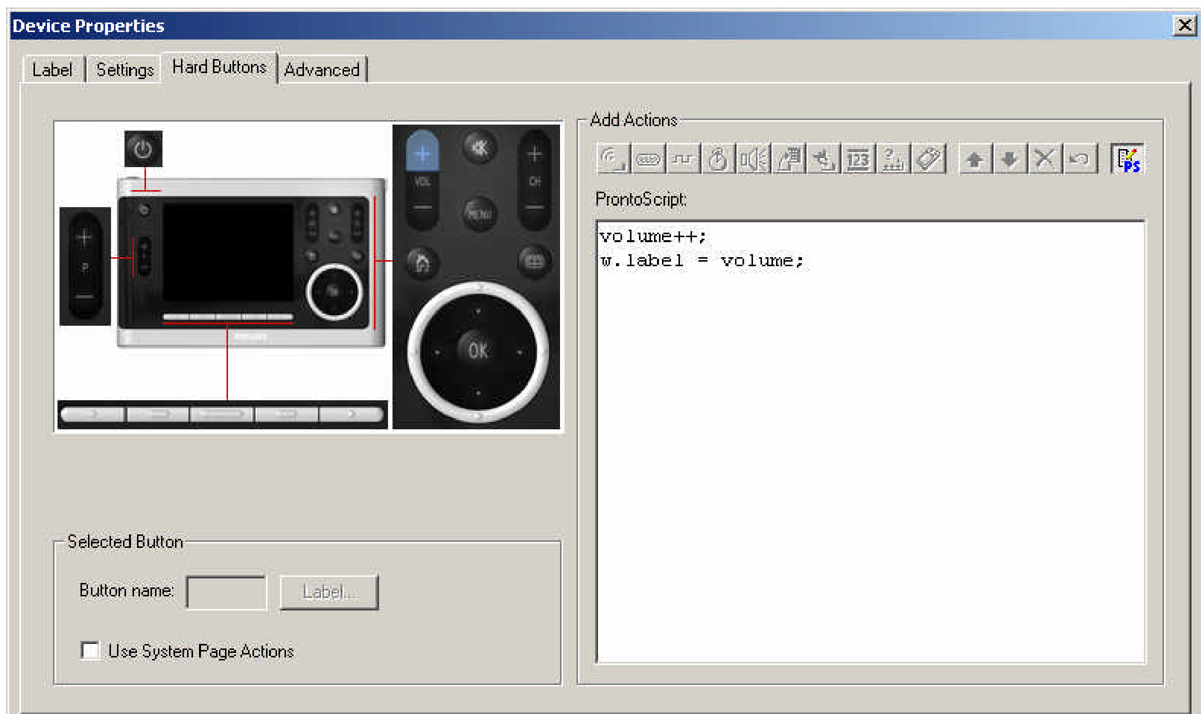
With the above page script, we retrieved a reference, `w`, to a panel with the tag `"VOLUME"`. Now we want to show a real dynamic value on it. Let's add some code to the page script to create a variable to contain this value:

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

When you download this to your control panel you will see that when 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 that 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, go to the Hard Buttons tab and select the vol+ button. Unselect the "Use System Page Actions" checkbox. Then click on the ProntoScript icon to show the script input field. Add the following code:



This little script increments our `volume` variable and updates the label of our volume panel with the new value. Note that we do not need to declare `volume` and `w` 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, add the following code 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. Was that easy or not?

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.

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 our 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?

Change the image

When you want to have a panel with dynamic graphics, you need to do some preparation. First, you need to collect the images you want to display and resize them to equal the panel size. Create a separate, hidden page in the same activity and attach each image 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 we can access those images:

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

This code copies the image of one of our resource panels to our 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.

In our example configuration above, in the button scripts for vol+ and vol-, replace the lines with:

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

by:

```
showVolume();
```

3.2. Buttons

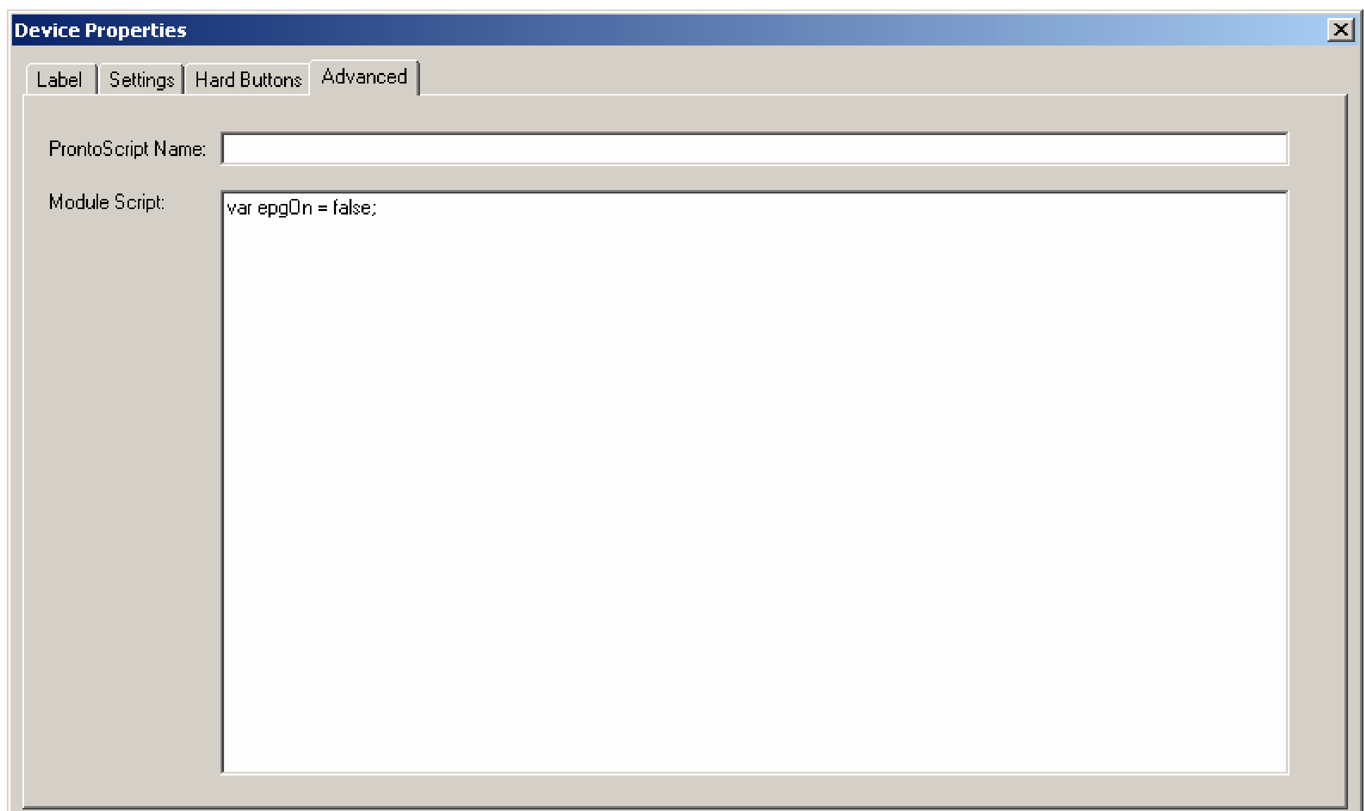
Buttons are put on a page to create a clickable area. So, you created a button, attached two images to indicate its released and pressed state and gave it a label to be displayed on it. And, of course, you attached actions to it. This is as far as you could go with the traditional Pronto buttons. Now let's see what we can do with it that we couldn't do before.

First, we give it a tag, for example "MY_BUTTON" and look it up in the page script:

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

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:



Then when the page with our button is displayed, we should initialize it properly in the page script:

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

This locates our button and then gives it the label "On" when `epgOn` equals `true`, and "Off" otherwise. Now in our button script we put some code to toggle the state and show it:

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

Note that we do not need to use the reference `w` here to get to the label because we are already inside the scope of our button object. For more information on button scope, refer to section 6.2.

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 label it `"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;
};
```

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 B: 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.

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. That's why we added the executeActions() method to widgets.

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

```
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 our 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:

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

Attention point:
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 absolutely wants the action list to be executed, it is advised to schedule a new executeActions() in the handling of this exception. See example in "2.5. Exceptions".

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:

```
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 10 minutes, so this should be enough for a coffee break. Note that the screen of the control panel looks frozen. Also the control panel does not respond to any key presses. So when someone calls on the phone there is no way to mute the stereo… Just ask them to call back after 10 minutes. Or maybe we should rewrite our script to be a bit more user friendly? Let's try the second timer mechanism:

5.2. Page timer

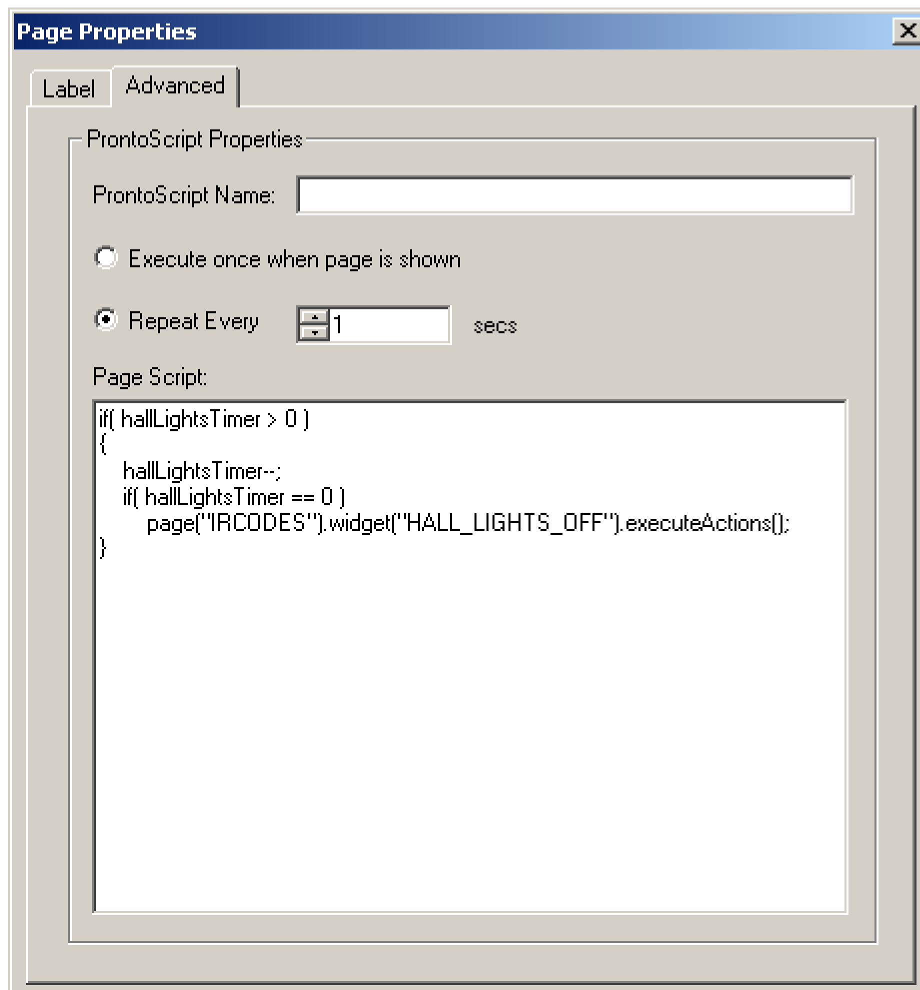
The editor allows you to mark a page script as repetitive. This feature can be used to count down until it is time to turn off the lights. First declare a counter in the activity script:

```
var hallLightsTimer = 0;
```

Then in the button script turn on the lights and start the timer by setting the counter. Let's program it to 10 seconds instead of 10 minutes for quicker testing:

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

And 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:



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 our 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". So we just need to define a function to turn off the lights, and call `scheduleAfter()` to trigger it, as shown in the following button script:

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

5.3. 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 screen of 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, that each in turn has 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

Now that we covered already numerous code 'snippets' and dropped a word or two on scope it is time to cover this subject in more detail.

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.

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.

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()` on activity level you can call it from the firm key scripts on that activity but also from one of the page scripts of that activity as well as from any button script on any one 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. That's why we also have:

System globals

There is a special facility for information that needs 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

We already gave some examples of activity and page scripts. Now it is time to discuss those 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()` on Activity level!)

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:

```
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.

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 we do not recommend it). Just add this line to the Home activity script:

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

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.

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.

Page label

In the editor you can define a label for every page. These labels could not be used before on the device. Now you can! Let's animate the activity label and the page label. Put this in the activity script:

```
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);
```

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;
}
```

Jump to other 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.

Multiple page jumps within 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 as 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 absolutely wants the action list to be executed, it is advised to schedule a new executeActions() in the handling of this exception. See example in "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 at 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 that, 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. So let's get the first serial port from the extender:

```
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 you can check against that:

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

Configuring the serial port

Now that we have the Serial object for the serial port that is connected to our receiver, we have to make sure it is configured with the same 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.

Sending and receiving

Now that we configured the serial port we can send a command to it to turn our 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 we can send a command and receive the response. 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.

Let's combine all above code snippets together in a button script that requests the volume and puts it on its label:

```
var e = CF.extender[0];
if( e == null )
{
    Diagnostics.log("Extender 0 is not defined");
}
else
{
    var 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);
    }
}
```

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. We already saw that blocking

the control panel is generally not a good idea. The proper way to do this is to define a callback function for the receiving of 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, our little function is called which sets the label.
You can also define a 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, but let's assume you configured the extender properly by now.

Getting the state

Now, imagine we want a panel on the page that should indicate the power state of a device. Let's tag it `"POWER_STATE"` and add a little page script to inquire the state of the input:

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

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

8.4. Relays

You can control an extender relay port as follows. First you get the corresponding Relay object:

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

And then you can retrieve the current state with `get()` and change it 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 the same 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 or so. 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. 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);
```

Similar as for 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.

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 "2.5. Exceptions" on handling exceptions, but let's first describe the case that everything goes well.
Once the connection is established, we can read and write to it. The following lines ask for the root directory in http format. 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 we are finished, we should close the connection:

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

We can combine the above code snippets in one button script to show the result on the button label when it is pressed:

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

Of course you should make sure you properly configured the wireless 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. In the next section we will show you how to avoid this.

Asynchronous operation.

When specifying false when constructing the TCPSocket, we get an asynchronous socket:

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

The next line looks identical as in 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 we cannot start writing to the socket, yet. So, the remainder of the script should be done when the connection is ready: in the `onConnect` callback function:

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

```
var socket = new TCPSocket(false);
var result = "";
socket.onConnect = function()
{
    write("GET /; HTTP/1.0\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 last thing we should add 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;
};
```

Chapter 10. Creating ProntoScript Modules

A ProntoScript module is an xcf 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.

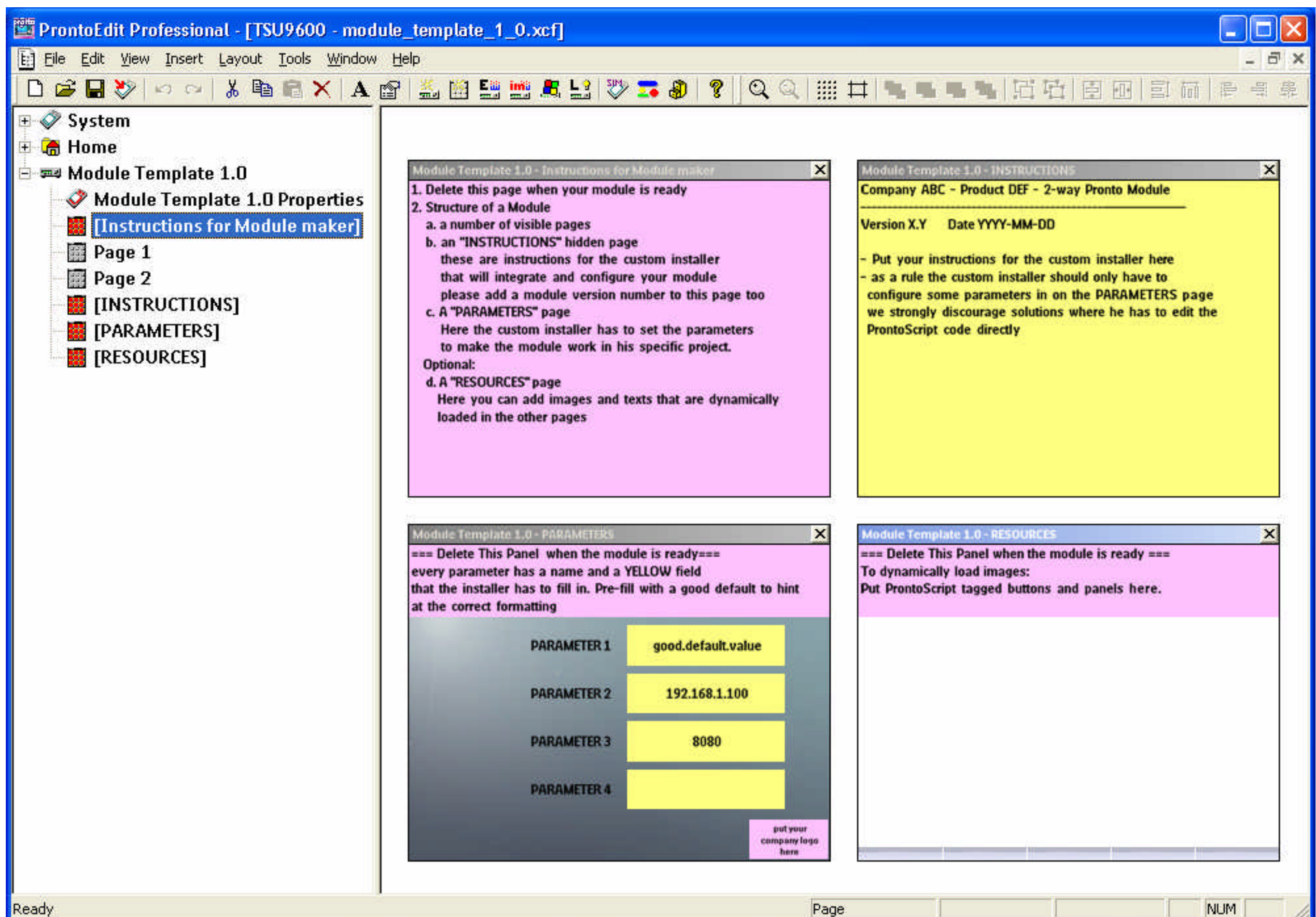
Merging XCFs to import activities

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 Configuration" option in the editor. This will add the activity at the bottom of the activity list.

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) to 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, we propose the following template as a standard with the SDK. We *strongly* encourage you to make use of it



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 pages with the same layout. Make sure the tags and labels of the parameters on this new page are numbered correctly.

Chapter 11. Exceptional scenarios

11.1. 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.

11.2. 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.

11.3. 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.

11.4. 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.

11.5. Script Exceptions

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

1	Exception	2	Description
3	"Failed"	4	The operation failed, e.g. reading from an extender serial port timed out.
5	"Not Implemented"	6	A class property or method was used that is currently not implemented.
7	"Not Available"	8	A class property or method was used that is not available.
9	"Insufficient internal memory available"	10	Not enough memory when reading from a TCP socket or setting a globa variable.
11	"Invalid name"	12	The name passed to System.getGlobal() or setGlobal() is not a proper string.
13	"Expected a function"	14	The specified callback is not a function.
15	"Expected an integer"	16	The parameter passed is not an integer.
17	"Expected a positive integer"	18	The specified Page.repeatInterval is negative.
19	"No argument specified"	20	Not enough arguments are passed to the class method.
21	"Not enough arguments specified"	22	Not enough arguments are passed to the class method.
23	"Argument is not a string"	24	Socket.connect expects a string as host name.
25	"Argument is not a function"	26	Activity.scheduleAfter expects a function.
27	"Argument is not an integer number"	28	A timeout must be an integer number.
29	"Argument is not a positive integer numer"	30	A timeout or the number of bytes to receive must be larger than 0.
31	"Argument is not an image"	32	Widget.setImage() is called with an invalid argument.
33	"Argument is not a boolean"	34	Input.match() is called with an invalid argument.
35	"Limit of simultaneous timers reached"	36	The maximum number of functions are already scheduleAftered
37	"Socket error"	38	A socket operation resulted in an error. E.g. a read() or write() failed.

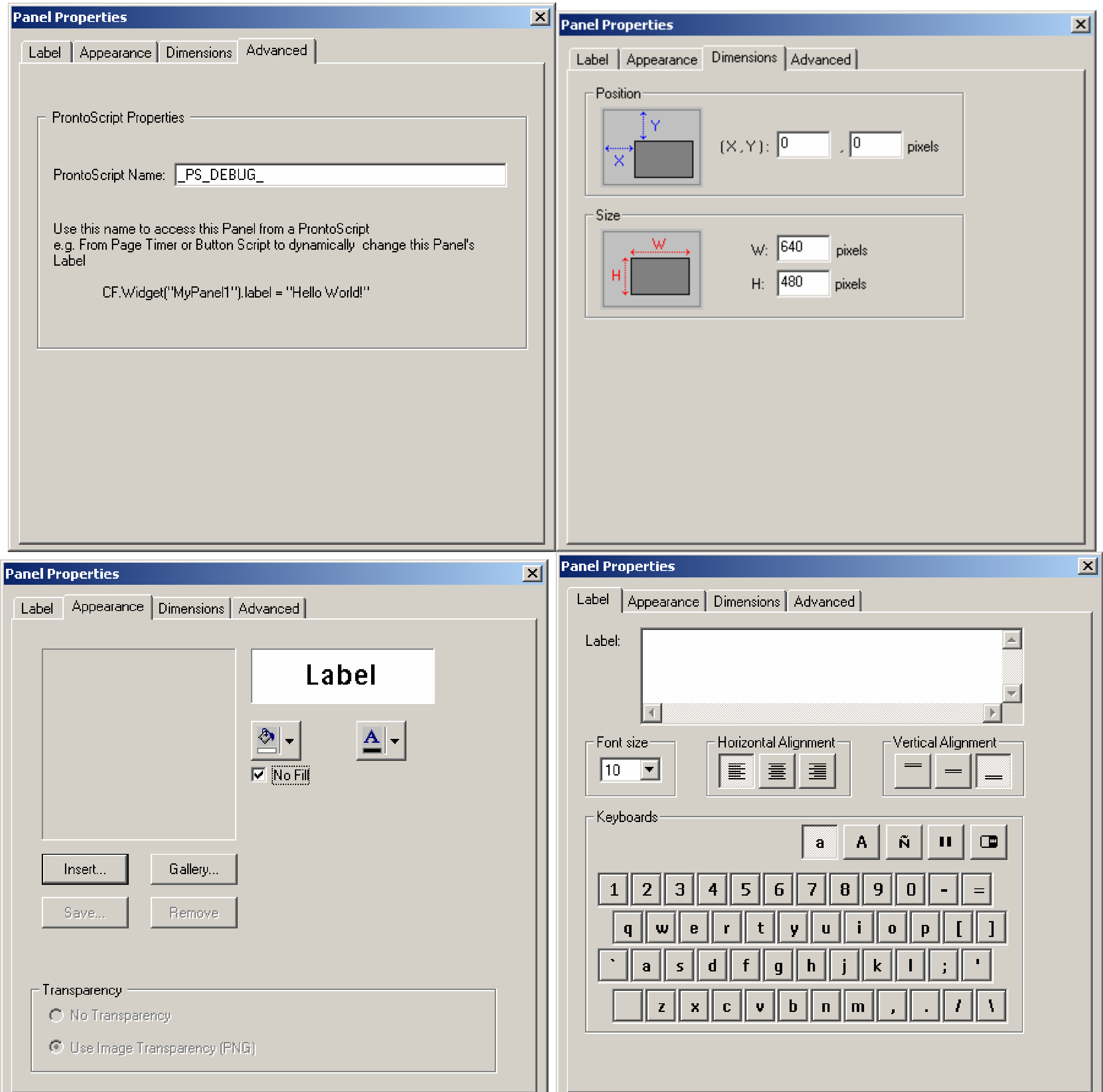
39	"Failed to connect"	40	Socket.connect() failed. Check your network settings.
41	"Maximum blocking read length exceeded"	42	You tried to read more than 65536 bytes from a synchronous socket.
43	"Maximum read length exceeded"	44	You tried to read more than 512 bytes from a serial port.
45	"ActionList Error"	46	Error during executeActions().

Chapter 12. 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.

12.1. Debug widget

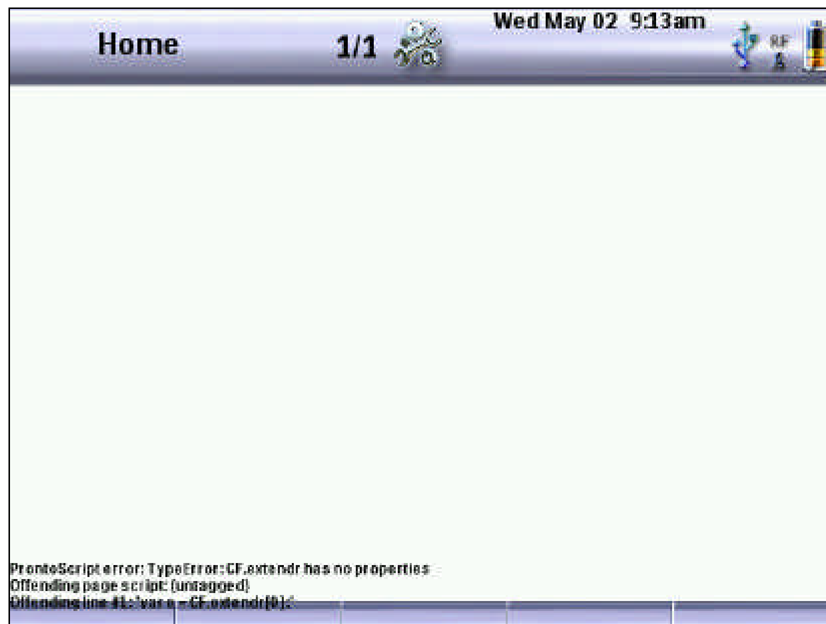
The first thing you can do is to create a debug widget on the page you want to debug. Create a panel with the tag `"_PS_DEBUG_"`. In the Dimensions tag, position it in the upper left corner and resize it to full screen. In the Appearance tab, check the No Fill box to make it transparent so you can still see and touch the other widgets on the page. Then in the Label tab, set the text alignment to bottom left.



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 we made a typo in our page script:

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

This will give the following output on the screen:

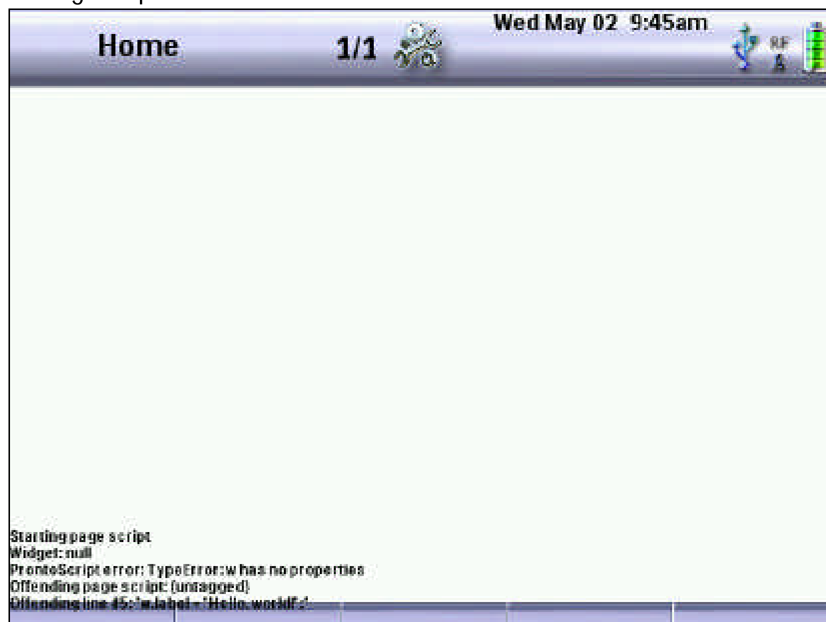


12.2. System.print()

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

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

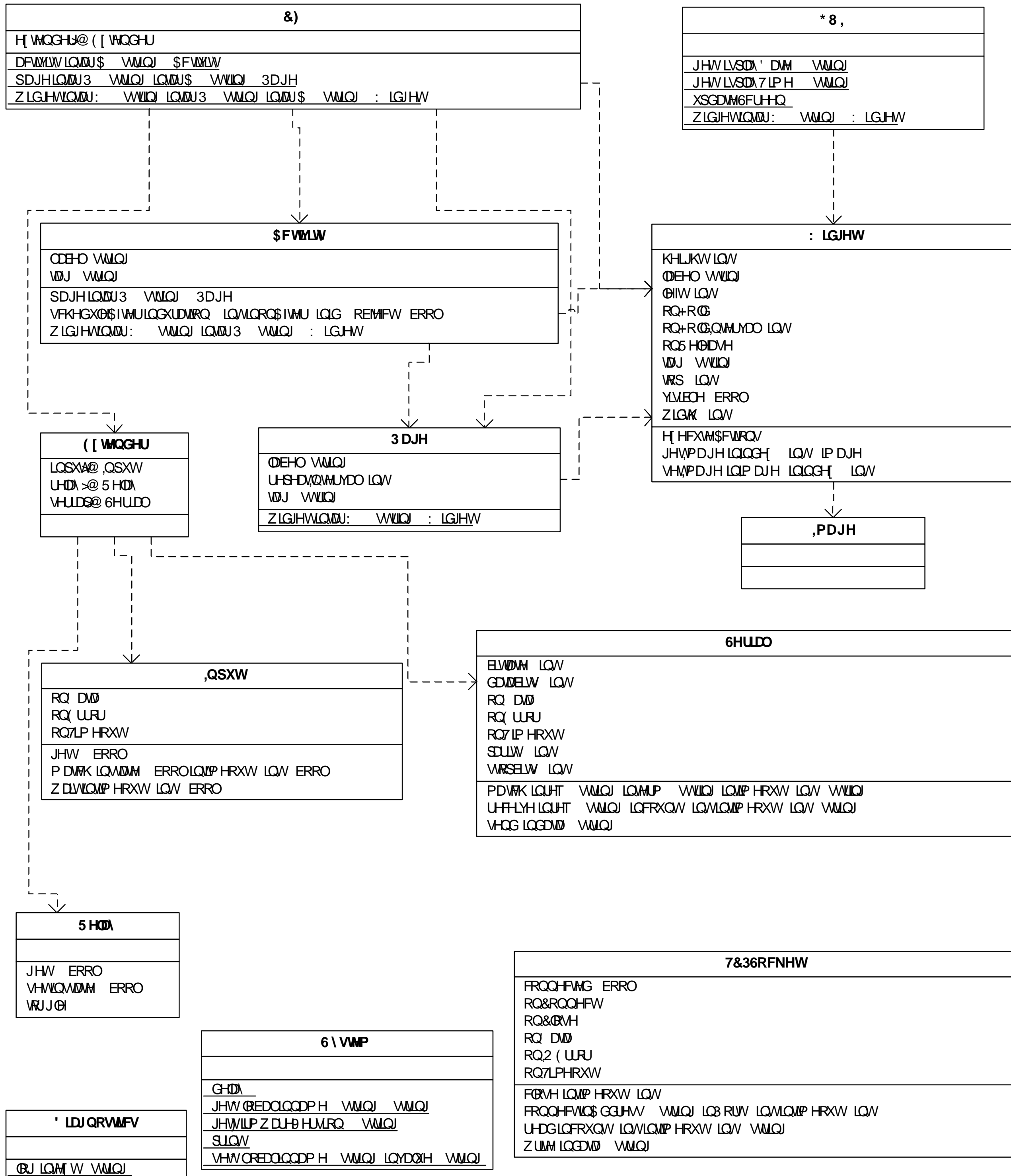
This will give the following output:



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

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 next diagram shows an overview of the classes:



The following sections list the available script object classes in alphabetical order.

A.1. Activity class

Description

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

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

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
Additional info :	-	

Class methods

scheduleAfter()

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	Anything	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 :	-		
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 that all control panel timers are paused while the control panel is asleep, postponing all pending function calls.		

Instance methods

page()

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 the Page Class chapter for a description of the return value. Refer to the Predefined Tags section for the applicable predefined page tags.		

widget()

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 the Widget Class section for detailed information on the return type. Refer to the Predefined Tags section for the applicable predefined page tags.		

A.2. CF class

Description

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

Class properties

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 the Extender Class section for an extensive description of its properties.	

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

Class methods

activity()

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 the Activity Class section for detailed information on the activity members. Refer to the Predefined Tags section for the tags to be used for the home and system activity.		
Example :	`CF.activity("DVD")`: returns the activity tagged "DVD". `CF.activity("")`: returns the current activity object.		

page()

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 tag *tagP* in the activity with tag *tagA*.
Exceptions :	- Argument is not a string		
Additional info :	Refer to the Page Class section for detailed information on the page class members. Refer to the Predefined Tags section for the tags to be used for the home and system page.		
Example :	`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.		

widget()

Purpose :	Searches the configuration file for a specific button, panel, hard key or firm key and returns the corresponding Widget class instance.		
Parameters :	tagW	String	Tag 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 page with the widget. If omitted, empty or null, the current activity is searched.
Return :	Widget		Class instance corresponding to the first matching widget in the specified page of the specified activity, or null if the widget is not found.
Exceptions :	- 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 the Widget Class section for detailed information on the page class members. Refer to the Predefined Tags section for the tags to be used for the home and system page.		
xamples :	`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 macro page in the current activity.		

A.3. Diagnostics class

Description

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 button + 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 properties

None

Class methods

log()

Purpose :	Add a message to the diagnostics log.		
Parameters :	s	String	The message text to be displayed.
Return :	-		
Exceptions :	-		
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 :	`Diagnostics.log("extender " + i + " does not respond");`		

A.4. Extender class

Description

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[]

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

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 the Relay Class section for more details on how to control the extender relays.	
Example :	`// Get relay port 0 on extender 0:` `var p = CF.extender[0].relay[0];`	

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 the Serial Class section for more details on the extender serial ports.	
Example :	`// Get access to serial port 0 of extender 0:` `var p = CF.extender[0].serial[0];`	

Instance methods

None

A.5. GUI class

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

Class properties
None

Class methods

getDisplayDate()

Purpose :	Get the control panel date.	
Parameters :	-	
Return :	String	Contains the date as shown in the Date status widget.
Exceptions :	-	
Additional info :	-	

getDisplayTime()

Purpose :	Get the control panel time.	
Parameters :	-	
Return :	String	Contains the time as shown in the Time status widget.
Exceptions :	-	
Additional info :	-	

updateScreen()

Purpose :	Force a screen update.
Parameters :	-
Return :	-
Exceptions :	-
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.

widget()

Purpose :	Search for a widget that is currently displayed on the screen. This also includes firm keys and hard keys and the widgets on the System page.		
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 the Widget Class section for detailed information on the return type.		

A.6. Image class

Description
This class represents an image in the configuration file. It cannot be instantiated directly. It is used when retrieving the image of 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
None

Instance methods
None

A.7. Input class

Description

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

Instance properties

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.
Additional info :	-	

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.	

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	
Additional info :	-	

Callback functions

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

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.
Additional info :	-		

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

onInputTimeoutCallback

Purpose :	Called when a timeout occurs during an asynchronous match() or wait() operation.
Parameters :	-
Additional info :	-

Instance methods

Note that 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!

get()

Purpose :	Get the value of the power sense input.		
Parameters :	-		
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.		

match()

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.		

wait()

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

Description

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

Instance properties

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.	

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.
Additional info :	-	

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.	

Instance methods

widget()

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 the Widget Class section for detailed information on the page class members.		
Examples :	`p.widget("RESULT")`: searches the widget tagged "RESULT" on page p.		

A.9. Relay class

Description

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

Instance properties

None.

Instance methods

get()

Purpose :	Inspect the actual value of a relay output.		
Parameters :	-		
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.		

set()

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.
Return :	-		
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.		

toggle()

Purpose :	Change the relay output state. If the relay was closed, it is opened. If it was open, it is closed.
Parameters :	-
Return :	-
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

Description

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.
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

Purpose :	Set the serial communication speed.	
Read/Write :	RW	
Value :	Integer	Valid values are: 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 57600 and115200 bits per second.
Additional info :	-	

databits

Purpose :	Set the number of data bits for the serial communication.	
Read/Write :	RW	
Value :	Integer	Valid values are 7 and 8.
Additional info :		

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.	

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.
Additional info :	-	

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.	

parity

Purpose :	Set the parity of the serial communication.	
Read/Write :	RW	
Value :	Integer	Valid values are: 0 (none), 1 (odd) and 2 (even).
Additional info :	-	

stopbits

Purpose :	Define he number of stop bits for the serial communication.	
Read/Write :	RW	
Value :	Integer	Valid values are 1 and 2.
Additional info :	-	

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.

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.		

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

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.		

Instance methods

Note that 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!

match()

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.		

receive()

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.		

send()

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.
Return :	-		
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

Description

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 properties

None

Class methods

delay()

Purpose :	Wait for a specific time. This blocks script execution during the specified time.		
Parameters :	duration	Integer	Duration of the delay in milliseconds.
Return :	-		
Exceptions :	- Not enough arguments specified - Argument is not an integer		
Additional info :	Note that the screen contents will not be refreshed during a delay. If this is desired, use *scheduleAfter* instead.		

getGlobal()

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		
Additional info :	-		

getFirmwareVersion()

Purpose :	Show the control panel firmware version.		
Parameters :	-		
Return :	String		Control panel firmware version, e.g. "TSU9600 V1.1"
Exceptions :	-		
Additional info :	-		

print()

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.
Return :	-		
Exceptions :	-		
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.		

setGlobal()

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.
Return :	-		
Exceptions :	- No argument specified - Argument is an invalid name - Insufficient internal memory available		
Additional info :	-		

A.12. TCPSocket class

Description

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

Class constructor

TCPSocket()

Purpose :	Create a new TCPSocket instance.		
Parameters :	Blocking	Boolean	Indicates if the new socket should be blocking (true) or not (false).
Return :	TCPSocket		A new TCPSocket class instance.
Exceptions :	-		
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.		

Instance properties

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.	

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.	

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.	

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.	

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.	

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:

onTCPSocketCloseCallback

Purpose :	Called when the socket is closed successfully.
Parameters :	-
Additional info :	-

onTCPSocketConnectCallback

Purpose :	Called when a connect() operation completes successfully on an asynchronous socket.
Parameters :	-
Additional info :	When the connect() is successful, the read() and write() operations can be used on the socket.

onTCPSocketDataCallback

Purpose :	Called when data is received on an asynchronous socket.
Parameters :	-
Additional info :	The callback function can retrieve the received data using the read() function.

onTCPSocketErrorCallback

Purpose :	Called when an error occurs on an asynchronous socket.

Parameters :	e	PanelError	An instance of the PanelError class for the error.
Example :	The error string can be retrieved by casting e to a string, e.g. `System.print( e );`		
Additional info :	-		

Instance methods

connect()

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 a synchronous connection.
Return :	-		
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. 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.		

close()

Purpose :	Terminate the connection.
Parameters :	-
Return :	-
Exceptions :	- Socket error
Additional info :	-

write()

Purpose :	Write data to a socket.		
Parameters :	s	String	The data to be transmitted. May contain binary data.
Return :	-		
Exceptions :	- Not enough arguments specified - Socket error		
Additional info :	The string data is queued for output on the network socket.		

read()

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

Description

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 that during script execution the screen is not updated, so any changes to widget properties will become visible after the script has finished. Refer to GUI.updateScreen() to force intermediate screen updates.
Because the Widget class is used to represents 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 also that 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

height

Purpose :	Determines the vertical size of the widget.	
Read/Write :	RW	
Value :	Integer	Range: 1…480
Applicable for :	Button, Firm key, Panel	
Additional info :	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.	

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	
Additional info :	-	

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.	

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 that 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 :	// 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; };	

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	
Additional info :	-	

Example :	// Button script showing a increasing speed counter:
	var counter = 1;
	var limit = 10;
	onHold = function() { label = count++; if (count == limit) { onHoldInterval /= 2; limit *= 2; } };	

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 of a button script that changes the button label when the button is released:
	`label = "Pressed";`
	`onRelease = function() { label = "Released"; };`

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.	

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.	

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	
Additional info :	-	

width

Purpose :	Determines the horizontal size of the widget.	
Read/Write :	RW	
Value :	Integer	Range: 1…640
Applicable for :	Button, Firm key, Panel	
Additional info :	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.	

Instance methods

executeActions()

Purpose :	Executes the action list attached to the button, if any.
Parameters :	-
Return :	-
Exceptions :	- ActionList error
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 that 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.

getImage()

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 :	Get an image from a panel in the gallery page of the current activity: `var img = CF.widget("IMAGE123", "GALLERY").getImage();`		

setImage()

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.
Return :	-		
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 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();
};
```

Appendix B: 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
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"
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 C: Pronto font

The following tables list the contents of the Pronto 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.

	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	{	|	}	~	

	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	A	a	A	a	A	a	C	c	C	c	C	c	C	c	D	d
0110	Ð	d	E	e	E	e	E	e	E	e	E	e	G	g	G	g
0120	G	g	G	g	H	h	H	h	I	i	I	i	I	i	I	i
0130	I	i	?	?	J	j	K	k	?	L	l	L	l	L	l	?
0140	?	L	l	N	n	N	n	N	n	?	?	?	O	o	O	o
0150	O	o	Œ	œ	R	r	R	r	R	r	S	s	S	s	S	s
0160	Š	š	T	t	T	t	T	t	U	u	U	u	U	u	U	u
0170	U	u	U	u	W	w	Y	y	Ÿ	Z	z	Z	z	Ž	ž	?
0190			ƒ													
02C0							ˆ	?								
02D0									?	?	˚	?	˜	?		
0390					?											
03A0										O						
03C0	p															
0410	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
0420	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
0430	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
0440	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
05D0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
05E0	?	?	?	?	?	?	?	?	?	'	"					

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2010				–	—				‘	’	‚		“	”	„	
2020	†	‡	•				…									
2030	‰									‹	›					
20A0													€			
2120			™													
2200			?													?
2210		?	-			/					v				8	
2220												?				
2240									~							
2260	?				=	=										
25C0											?					

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
F020		?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F030	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F040	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F050	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F060	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F070	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F080	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F090	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F0A0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F0B0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F0C0	?	?	?	?	?	?	?	?	?		?	?	?	?	?	?
F0D0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F0E0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
F0F0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
FB00		?	?													

Further reading

[Flanagan] David Flanagan. Copyright © 2006, 2002, 1998, 1997, 1996, O'Reilly Media, Inc. ISBN 0-596-10199-6. O'Reilly Media, Inc., *JavaScript: The Definitive Guide*, Fifth Edition.

Note: We strongly encourage you to get a copy of this 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.

[JSLint] http://www.jslint.com/: provides tools to check your script for errors.

[Mozilla] http://developer.mozilla.org/en/docs/JavaScript: contains a very extensive reference and a guide on the Core JavaScript 1.6, as well as a "re-introduction to JavaScript" as they call it.

Glossary

CF	- Configuration file
Editor	- The ProntoEdit Professional program to create Pronto configurations
Firm key	- The five physical buttons below the control panel screen
Hard button	- The physical buttons of the control panel
Tag	- The ProntoScript Name that can be specified in the editor for a widget
Widget	- Graphical element on the screen. A button, panel or hard button