PAWN Scripting in Life OS v1.0

Welcome to the community of developers who are taking UGOBE Life Forms™ to their next steps…literally. This document is the beginning of how you can expect UGOBE™ to work together with you, through documentation, Q&A, Forums chat and a full-fledged Development Kit in 2008.

Overview

What is PAWN? PAWN is a simple, typeless, 32-bit extension language with a C-like syntax. If you know C, you know PAWN! PAWN was designed for execution speed, stability, simplicity and a small footprint. The PAWN language has been built into the Pleo™ firmware to allow arbitrary code execution, accessing the functionality of the Pleo firmware. See http://www.compuphase.com/PAWN/PAWN.htm or http://code.google.com/p/PAWNscript/ for more details on PAWN.

Audience
This document is intended for experienced PAWN programmers as well as experienced C programmers who would like to learn about PAWN. It is not intended for people with no programming experience.

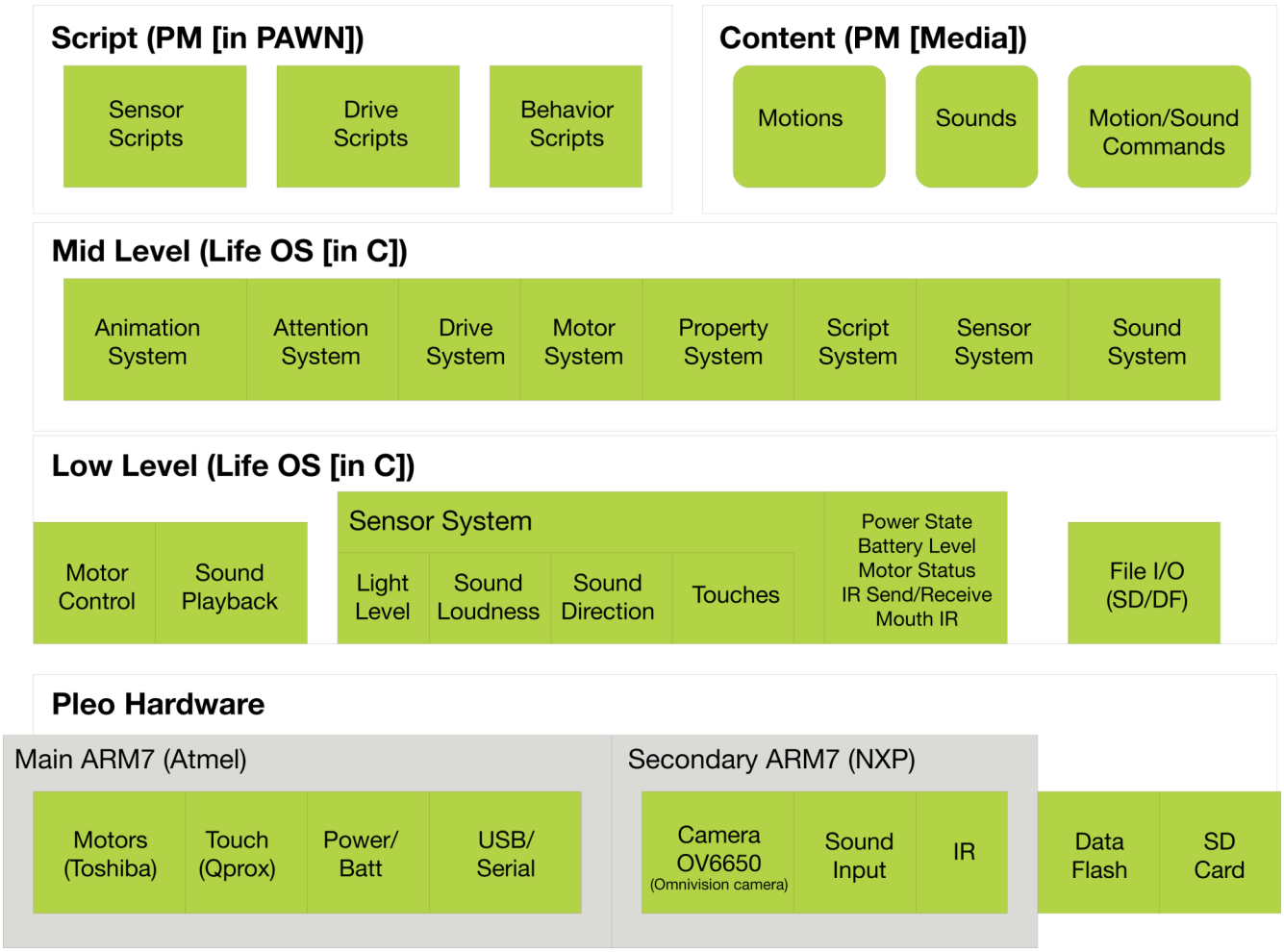
Objective
With this document, programmers should be able to gain a general understanding of the process by which applications are written for Pleo.

Pleo Developers Kit (PDK)
UGOBE plans to release a full PDK later this year that will provide tools to assist in creating applications for Pleo. Please join the Pleo Updates mailing list at www.pleoworld.com to keep posted on our plans.

Support
Technical questions? Please join the Pleo Community Forum and participate in the technical thread UGOBE Tech Talk. Visit www.pleoworld.com to join!

Background

The graphic below shows the constituent parts of the Pleo (Life OS) software. Please note that the initial PDK release may not necessarily give access to all of these pieces. Our initial focus will be on the sound, motion, command, sensor and property systems, along with the basic operating system routines. Other higher-level components will be exposed and documented in future releases.



Life OS Applications

A Life OS application is a collection of a number of different resources. These resources include sounds, motions, commands, scripts and properties. These are described here.

Sounds

The build tools use the built-in Python audioops module to process incoming WAV files into raw PCM or ADPCM formatted USF files (UGOBE Sound File (USF) format.). The Sound System can currently play 11k, 8-bit mono sound data.

Motions

Motions are resources that contain data on how and when to move joints in Pleo. Internally we animate using a 3ds Max model whose animation data we export using a custom MaxScript. This results in a CSV file, with a special header with some meta-information about the motion – frame rate, duration, which joints are used, etc. This CSV file is processed into a UGOBE Motion File with a special Python script.

Commands

Commands are tables of motions, each with associated state information, which describes when that specific motion can be played. The Command playback system (Animation System) gives Pleo much of his expressiveness and variability and at any particular instant in time, there may be multiple motions to choose from. The Animation System will choose a motion from among the group.

Scripts

The backing code of an application is written in the PAWN scripting language, as is the majority of the high-level behavior in Pleo. The scripts are compiled against the Pleo include files, which detail all the native functions available in the Pleo firmware. Scripts are processed using the PAWN compiler and a post-processing tool to arrange the code into pageable code blocks. For the majority of Pleo applications, there are three PAWN VMs available at run-time to execute various scripts. These are detailed below. Note that PAWN script files are usually suffixed by a .p extension.

Build Tools

Internally we use Python scripts to process the majority of resources. We describe an application and its associated resources via a UGOBE Project File, or UPF. This is an XML formatted file that lists all resources, and sets various build options for those resources. These tools will be demonstrated below. Note that Python files are usually suffixed by a .py extension.

PAWN Virtual Machines (VMs)

In Life OS 1.0, we have implemented four different instances of the PAWN VM. We did this in order to support a kind of multi-tasking without having to make extensive changes to the PAWN language itself. The four VMs we refer to as: sensor, main, behavior and user. These are detailed below.

Note that every script can contain an `init` and a `close` function. If a script contains an `init` function, it will be called just after the script has been successfully loaded. And the `close` function will be called just before the script unloaded. Note that both of these functions are optional.

Sensor VM

The Sensor VM is the VM designed to execute the sensors.amx scripts, which contains the on_sensor callback function. This function is called on each sensor "trigger". A trigger is when a sensor has changed in a meaningful way. For example, a touch sensor would trigger on a "touch" or a "release", but a sound loudness sensor would trigger only when there is a sound loudness change of greater than X amount – the X value being programmable at run-time.

Main VM

The Main VM is the VM designed to run the main.amx script. This VM will look for a public function named `main`. If it exists, it will be executed. The VM does not expect this main function to return; that is, there is typically a `while (true)` or `for (;;)` loop within the main function, preventing it from ever returning. But if the main function does return, the Main VM will simply call the `main` function again. This script is typically used to track some global property or properties – set in response to some sensor trigger in the sensors script – and then execute another script in the Behavior VM.

Behavior VM

The Behavior VM is the VM designed to run any other arbitrary script. Scripts are loaded with a vm_exec native function, which takes a script ID or name as a parameter. The VM will load this script, and look for a public `main` function. If there is a public `main` function, it will be called. The current running status of the script in the Behavior VM will be stored in a property named property_script_status. When the main function in the script exits, the `close` function – if any – will be called. And when the `close` function exits, the property_script_status will be set to "done". The main script will typically poll this property, and when it is set done, it can load some other script.

User VM

The User VM is the VM designed to run a script outside the current application context. The current application context usually means any scripts running in the Sensor, Main and Behavior VM. When an application is loaded, it may use these three VMs. But the script running in the User VM can run whether an application is running or not, and does not automatically start or stop with an application. This allows some very interesting utility scripts to be written, from debug scripts to installer scripts.

Life OS PAWN native interface

The API exposed to PAWN from the Life OS firmware is referred to as the native interface. We have broken up the exposed functionality into related groups, which roughly match the application and OS services implemented in the Life OS. Each group, or module, API is defined in a PAWN include file, using the .inc file suffix. The modules include:

- Animation: functions dealing with Commands
- Application: functions dealing with loading and unloading applications
- File: File IO functions for DataFlash and SD Card
- Joint: functions dealing with moving individual joints
- Log: functions dealing with log, or monitor, output
- Motion: functions that deal with motion playback
- Property: functions dealing with the Property system
- Resource: functions dealing with resource (URF) files
- Script: functions dealing with the PAWN VMs
- Sensor: functions dealing with sensors
- Sound: functions dealing with sound playback
- String: functions dealing with PAWN strings
- Time: functions dealing with time
- Util: useful utility functions

Note that there are additional modules in Life OS 1.0 available for use by the Pleo personality module (PM) however, these are not being described here.

Pleo Application Example

Overview

This example will demonstrate a very simple application that will play a sound in response to a sensor touch. This application consists of one script and one sound file. This will give you a rough idea of how Pleo applications are built, using the Python-based build tools that will be included in the PDK. Other tool sets may be used in future to build Pleo applications.

Prerequisites

Note you will need to have a version of Python installed before building with the Python scripts in the PDK. We recommend Python 2.5, since it includes the ElementTree module we use to parse the XML-based UPF file. Previous versions should work if you install the ElementTree module separately.

Preparation

Included in the PDK will be a template project. This project includes a skeleton of a basic Pleo application. We will use this template as a base to create our new application. Follow these steps to create a new work area:

1. Create a new folder named “touch_test” under the examples folder in the pdk folder. Our template project files assume relative paths, so it is best to keep all the examples/projects at the same level.
2. Copy the sensors.p PAWN file from the template folder to your newly created touch_test folder.
3. Copy and rename the template.upf file to touch_test.upf in your touch_test folder.
4. Create a sounds folder under your touch_test folder.
5. Place a WAV file in the sounds folder you want to play in response to the touch sensor. This WAV file should be in 11k, 8-bit, mono PCM format. There is an example sound in the template/sounds folder, if you wish to experiment with that.

You should now have a folder that looks like this:

```
/pdk
        /examples
                /touch_test
                        /sounds
                                sample.wav
                        sensors.p
                        touch_test.upf
```

Now we need to edit the sensors.p file to add the code that responds to the touch sensor. First, let's review the code. It is as follows:

```
 1 //
 2 // Very simple sensors.p example. Add code to on_sensor for those
 3 // sensors you would like to respond to.
 4
 5 //
 6 // save space by packing all strings
 7 #pragma pack 1
 8
 9 #include <Log.inc>
10 #include <Script.inc>
11 #include <Sensor.inc>
12
13
14 public init()
15 {
16     print("sensors:init() enter\n");
17
18     print("sensors:init() exit\n");
19 }
20
21 public on_sensor(time, sensor_name: sensor, value)
22 {
23     new name[32];
24     sensor_get_name(sensor, name);
25
26     printf("sensors:on_sensor(%d, %s, %d)\n", time, name, value);
27
28     switch (sensor)
29     {
30     }
31
32     // reset sensor trigger
33     return true;
34 }
35
36 public close()
37 {
38     print("sensors:close() enter\n");
39
40     print("sensors:close() exit\n");
41 }
```

At line 7, we add a PAWN option to “pack” strings in the resultant AMX file. By default in PAWN will store strings into arrays of cells. A cell in our case is a 32-bit value, so each string would take up a large amount of memory. Added the pack option will put four characters into each cell.

At lines 9-11, we call out the Pleo `include` files that contain the native function prototypes that we need in this script. These `include` files can be found in the pdk/include folder. In this case, Log.inc defines the `print` and `printf` functions, Script.inc includes the proper

prototypes for our `init`, and `close` functions and the Sensor.inc defines the prototype for on_sensor and defines the functions sensor_get_name and pulls in the proper definitions for the sensor_name enumeration, located in the pdk/include/pleo folder.

Lines 14-19 define an initialization function that will be called by the Pleo firmware when this script is loaded. This function is optional – if it is not present, the firmware will continue on.

Lines 21-34 define the on_sensor function that is called each time a sensor is "triggered". A trigger may be a touch sensor press, a touch sensor release, a sound loudness change, etc. We will explain this function in more detail below when we add some code to it.

Lines 36-41 define a close function that is called before the script is unloaded. Like the `init` function, this is an optional function.

Some general notes on this script:

Note that the three functions defined here are declared "public" This is a PAWN keyword that will add that function name to a list of functions that may be called from the firmware. You can define your own local functions, which would not use the public keyword.

Note there are no integral types used in this script. All variables in PAWN are cells, arrays of cells, or enums. The one enum in this script is sensor_name, which is a collection of sensor Ids whose values are shared between the firmware the PAWN script.

Note on line 23 how a variable is declared using the new keyword. This tells the PAWN compiler to allocate space for that variable in the data section of the script. In this case, it is an array of cells. This results in the reservation of 128 bytes in the resultant AMX data section.

New Code

Now we want to add the additional code necessary to play a sound in response to a touch sensor. Follow these steps:

At line 12, add the following:

```
#include <Sound.inc>
```

This include file defines the sound playback natives that will be used to play sounds

At line 13, add the following:

```
#include "sounds.inc"
```

This will pull in an `include` file that defines the sound names available for playback. The sounds.inc file now gets created when we build the project, as described below.

After line 29, include this code:

```
case SENSOR_BACK:
     if (value == 0)
     {
          sound_play(snd_sample);
     }
```

You should use the name “snd_<name>”, where <name> is the base file of the WAV you placed in the sounds folder.

The resultant code should look this this:

```
//
// Very simple sensors.p example. Add code to on_sensor for those
// sensors you would like to respond to.
//

// save space by packing all strings
#pragma pack 1

#include <Log.inc>
#include <Script.inc>
#include <Sensor.inc>
#include <Sound.inc>

#include "sounds.inc"


public init()
{
    print("sensors:init() enter\n");

    print("sensors:init() exit\n");
}

public on_sensor(time, sensor_name: sensor, value)
{
    new name[32];
    sensor_get_name(sensor, name);

    printf("sensors:on_sensor(%d, %s, %d)\n", time, name, value);

    switch (sensor)
    {
    case SENSOR_BACK:
      if (value == 0)
      {
           sound_play(snd_growl);
      }
    }

      // reset sensor trigger
    return true;
}
```

```
public close()
{
    print("sensors:close() enter\n");

    print("sensors:close() exit\n");
}
```

Building

Now it is time to build the project. The end result of the build will be a pleo.urf file, which can be written onto an SD Card, inserted into Pleo, and executed.

Note we are using the Python build tools that we developed internally for Pleo. It is based on the command line. That is, no fancy project or code editors – yet!

First, we must edit the touch_test.upf file, which was simply copied from the template.upf file. Here is what it looks like initially:

```
 0 <ugobe_project name="template">
 1
 2   <options>
 3     <set name="top" value="../.."/>
 4     <include value="./include:${top}/include"/>
 5     <tools>
 6      <PAWN value="${top}/bin/PAWNcc %i -O1 -S64 -v2 -C- %I -o%o"/>
 7      <block value="${top}/bin/pleocc -b512 -v %i"/>
 8     </tools>
 9     <directories>
10       <build value="build"/>
11       <include value="include"/>
12     </directories>
13     <umf value="3" />
14     <block />
15     <folders />
16   </options>
17
18   <set-default name="MEDIA" value="."/>
19
20   <set name="SOUNDS" value="${MEDIA}/sounds"/>
21   <set name="MOTIONS" value="${MEDIA}/motions"/>
22   <set name="SCRIPTS" value="${MEDIA}/scripts"/>
23
24   <resources>
25
26     <!-- Sounds -->
27     <sound path="${SOUNDS}/growl.wav"/>
28
29     <!-- Motions -->
30     <motion path="${MOTIONS}/bow.csv"/>
31
32     <!-- Scripts -->
33     <script path="sensors.p" />
34     <script path="main.p" />
35
36   </resources>
37 </ugobe_project>
```

Some descriptions of the parts follows.

Lines 2-16 define options that are passed to the build tools. Line 2 is a macro definition to point to the root of the pdk folder. This allows other commands to use this macro. Line 4 defines the include path to be used in the tools section. Line 6 defines the command line options to be passed to the PAWN compiler. Line 7 defines the command line options to pass to the Pleo PAWN post-processor. Line 10 defines where to put the resultant files. Line 11 defines where to look for local `include` files. Line 13 says to use version 3 (vectored) UMF files. Line 14 says to code block the resultant AMX files. And line 15 says to put the sounds, motions and commands into folders in the destination folder.

Lines 24-36 define the actual resources that make up this application. The project tool will enumerate through this list, building each resource. Then it will combine all of those resources into the final pleo.urf file.

Changes

We need to make the following changes for our touch_test.

- On Line 0, change the name of the project to “touch_test”.
- Remove line 30, since we do not use any motions in this sample.
- Remove line 34, since we do not use a main.p script, but only a sensors.p script.

The resultant UPF file should look like this.

```
<ugobe_project name="touch_test">

  <options>
    <set name="top" value="../.."/>
    <include value="./include:${top}/include"/>
    <tools>
     <PAWN value="${top}/bin/PAWNcc %i -O1 -S64 -v2 -C- %I -o%o"/>
     <block value="${top}/bin/pleocc -b512 -v %i"/>
    </tools>
    <directories>
      <build value="build"/>
      <include value="include"/>
    </directories>
    <umf value="3" />
    <block />
    <folders />
  </options>

  <set-default name="MEDIA" value="."/>

  <set name="SOUNDS" value="${MEDIA}/sounds"/>
  <set name="MOTIONS" value="${MEDIA}/motions"/>
  <set name="SCRIPTS" value="${MEDIA}/scripts"/>

  <resources>

      <!-- Sounds -->
    <sound path="${SOUNDS}/growl.wav"/>
```

```
      <!-- Motions -->

      <!-- Scripts -->
    <script path="sensors.p" />

  </resources>
</ugobe_project>
```

Now to build it!

Open a Command Prompt (Windows), bash shell (cygwin or Linux) or a Terminal (OSX). Change directories to the touch_test project folder, and type the following:

```
python ../../bin/ugobe_project_tool.py touch_test.upf rebuild
or
python ..\..\bin\ugobe_project_tool.py touch_test.upf rebuild
```

You will see a lot of output from the build tools, like so>

```
*** Cleaning ***
  Removing include/sounds.inc
  Removing sounds.xml
  Removing include/scripts.inc
  Removing scripts.xml
  Complete Clean: Removing build directory 'build'

*** Prepocessing ***
  Updating enumeration XML 'sounds.xml'
  Creating enumeration 'include/sounds.inc'
  Updating enumeration XML 'motions.xml'
  no data for motions
  Updating enumeration XML 'commands.xml'
  no data for commands
  Updating enumeration XML 'scripts.xml'
  Creating enumeration 'include/scripts.inc'
  Updating enumeration XML 'user_properties.xml'
  no data for user_properties

*** Processing ***
  Converting ./sounds/growl.wav to build/sounds/4096.usf adpcm:0 pitch:1
  @ ..\..\bin\PAWNcc sensors.p -O1 -S64 -v2 -C-  -iinclude -i..\..\include -obui
ld/sensors.amx
PAWN compiler 3.2.3664                  Copyright (c) 1997-2006, ITB CompuPhase

..\..\include\Sound.inc(51) : warning 213: tag mismatch
..\..\include\Sound.inc(59) : warning 213: tag mismatch
Header size:            176 bytes
Code size:              460 bytes
Data size:              128 bytes
Stack/heap size:        256 bytes; estimated max. usage=43 cells (172 bytes)
Total requirements:    1020 bytes

2 Warnings.
  block compiling 'build/sensors.amx'
  @ ..\..\bin\pleocc -b512 -v build/sensors.amx
Size increase         = 348 bytes
```

```
Block size            = 512
Pad size              = 512
Number of blocks      = 1
Number of adjustments = 1

*** Writing build.urf ***
Bad version element in UPF or cannot get Subversion revision. Using 0
Version is 0
Build Time is 1200598476 (Thu Jan 17 15:34:36 2008)
writing build/sounds/4096.usf (growl) at 0x200L
writing build/sensors.amx (sensors) at 0x4200L
writing UGSF toc at 0x4800L
writing UGMF toc at 0x4830L
writing UGCF toc at 0x4838L
writing  AMX toc at 0x4840L
writing PROP toc at 0x4870L
  URF file fits: 18560 of 3649536. 3630992 free

Adler32 crc is B9DA7A93
build time: 0.516000 sec
```

If the build was successful, you will have a pleo.urf file in the build directory off the touch/test folder. Copy this file to an SD Card, insert into Pleo (while turned off), and then turn on Pleo. Touch his back, and you should hear the sound that you added. Congratulations!